# O retorno do cangaceiro

# JavaScript

De padrões a uma abordagem funcional


Casa do Código

SÉRIE CAELUM

FLÁVIO ALMEIDA

*Edição*

Adriano Almeida
Vivian Matsui

*Revisão*

Bianca Hubert
Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

## SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

# ISBN

Impresso e PDF: 978-85-94188-81-6

EPUB: 978-85-94188-82-3

MOBI: 978-85-94188-83-0


Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

# AGRADECIMENTOS

Quero agradecer a minha filha e a minha esposa, que sempre foram a maior motivação para que eu concluísse todos os meus empreendimentos, inclusive este livro.

Não poderia deixar de fora da lista de agradecimentos Vivian Matsui por todo o suporte durante a criação desta obra, inclusive suas dicas e sugestões que sempre visavam tornar o material ainda mais esclarecedor. Agradeço também à cangaceira JavaScript Loiane Groner, que aceitou sem hesitar o meu convite para escrever o prefácio.

Por fim, um agradecimento especial para Celso Silva, que indiretamente contribuiu com esta obra, permitindo que eu abrisse minha mente para novas ideias e experiências.

# PREFÁCIO POR LOIANE GRONER

O JavaScript é uma linguagem que está em constante evolução. Com as ferramentas disponíveis hoje, podemos usar diferentes paradigmas e até mesmo usar mais de um paradigma em um mesmo projeto.

O paradigma funcional está cada vez mais presente em frameworks modernos. O retorno do cangaceiro JavaScript é um livro que nos leva a uma jornada e nos ensina conceitos da Programação Funcional e conceitos avançados do JavaScript de maneira bem fluida, fazendo com que a gente aprenda conceitos mais difíceis sem perceber. Após os conceitos iniciais, o livro nos ensina os conceitos da Programação Funcional reativa, através de exemplos práticos com a criação da nossa própria biblioteca e operadores. Por fim, aprendemos os padrões mais utilizados e técnicas que podemos usar para melhorar nossos projetos.

O autor aborda conceitos e técnicas que são essenciais para pessoas desenvolvedoras de front-end, podendo também ser utilizados no back-end e até mesmo em outras linguagens. Com o conhecimento adquirido neste livro, será possível se aventurar em terras ainda mais avançadas e frameworks que possuem demanda no atual mercado de trabalho.

Jogue-se no retorno dessa aventura e se divirta com os códigos!

# SOBRE O AUTOR

Figura 1: Flávio Almeida

Flávio Almeida é desenvolvedor e instrutor na Caelum, empresa na qual ministrou mais de 5.000 horas de cursos de tecnologia Java e front-end para indivíduos e empresas como Petrobras, SERPRO, BNDS, Ministério da Justiça, Globo.com, Banco Central, entre outras entidades públicas e privadas. Também é instrutor na Alura, publicando mais de 20 treinamentos para esta plataforma de ensino online. Autor do *best-seller Cangaceiro JavaScript: uma aventura no sertão da programação* e do livro *Mean: Full stack JavaScript para aplicações web com MongoDB, Express, Angular e Node*, possui mais de 15 anos de experiência na área de desenvolvimento. Bacharel em Informática com MBA em Gestão de Negócios em TI, tem Psicologia como segunda graduação e procura aplicar o que aprendeu no desenvolvimento de software e na educação.

Atualmente, foca na linguagem JavaScript e no desenvolvimento de Single Page Applications (SPA), tentando

aproximar ainda mais o front-end do back-end através da plataforma Node.js.

Já palestrou e realizou workshops em grandes conferências como QCON e MobileConf, e está sempre ávido por novos eventos.

# INTRODUÇÃO

> *"O cangaceiro deve ser desconfiado e ardiloso como raposa, ter agilidade do gato, saber rastejar como cobra e desaparecer como o vento."* - Lampião

## A QUEM SE DESTINA O LIVRO

Este livro destina-se àqueles que desejam aprimorar a manutenção e legibilidade de seus códigos, aplicando o paradigma funcional e padrões de projetos.

Procurei utilizar uma linguagem menos rebuscada para tornar este livro acessível a um amplo espectro de desenvolvedores. No entanto, é necessário que o leitor já tenha conhecimento da linguagem JavaScript para um melhor aproveitamento.

Além de conhecer a linguagem JavaScript, é recomendado que o leitor já tenha um conhecimento, mesmo que básico, de módulos e da API Fetch. Para diminuir o *gap* entre o leitor e o livro, será realizada uma revisão desses dois tópicos. Aliás, ambos já foram abordados no livro *Cangaceiro JavaScript: uma aventura no sertão da programação* deste mesmo autor (https://www.casadocodigo.com.br/products/livro-cangaceiro-javascript).

## SOBRE A NOSSA JORNADA

O leitor será introduzido gradativamente aos jargões e técnicas do paradigma funcional aplicados à linguagem JavaScript, além de padrões de projetos que visam melhorar a manutenção e legibilidade do código.

Você aprenderá várias técnicas que farão parte do seu arsenal para resolver problemas comuns do seu dia a dia como programador front-end utilizando apenas *vanilla* JavaScript.

Por fim, grande parte do que você aprenderá servirá como base para entender a motivação e dominar outras bibliotecas e frameworks do mercado.

Vejamos a seguir a infraestrutura necessária para o projeto do livro.

## INFRAESTRUTURA NECESSÁRIA

Como utilizaremos o sistema de carregamento nativo de módulos JavaScript nos navegadores, é necessário que o leitor utilize navegadores compatíveis. A boa notícia é que, na data de publicação deste livro, o carregamento de módulos nativos é suportado pelos principais navegadores.

O autor testou todo o código escrito no livro no **Google Chrome 63**. É possível verificar o suporte ao carregamento de módulos através da tag `<script>` pelos navegadores em (https://caniuse.com/#search=modules).

Precisaremos que a plataforma **Node.js** esteja instalada para que o servidor local disponibilizado com o download do projeto seja executado. O Node.js pode ser baixado no endereço https://nodejs.org/en/. Durante a criação deste projeto, foi utilizada a versão 8.1.3. Mas não há problema em baixar versões mais novas, contanto que sejam **versões pares**, as chamadas versões LTS (*Long Term Support*).

## DOWNLOAD DO PROJETO

Há duas formas de baixar o projeto base deste livro. A primeira é clonar o repositório:

https://github.com/flaviohenriquealmeida/cangaceiro2/.

Você também pode baixá-lo no formato `zip` através do endereço:

https://github.com/flaviohenriquealmeida/cangaceiro2/archive/master.zip/.

No entanto, caso o leitor tenha optado pela versão `zip`, depois de descompactá-la, precisará renomear a pasta `cangaceiro2-master` para `cangaceiro2`, de modo a haver uma

paridade da estrutura abordada no livro com o projeto baixado.

O projeto disponibiliza um servidor Web que torna acessível nossa aplicação para o navegador. Além disso, ele possui *endpoints REST* para que sejam consumidos através do nosso código, dessa maneira, criando situações mais próximas do que enfrentamos no dia a dia.

**Visual Studio Code (Opcional)**

Durante a criação do projeto, foi utilizado o Visual Studio Code (https://code.visualstudio.com/download), um editor de texto gratuito e multiplataforma disponível para Windows, Linux e MAC. Sinta-se livre para escolher o editor que você preferir.

## BEM COMEÇADO, METADE FEITO

Com o projeto baixado e toda a infraestrutura necessária disponível, podemos dar início à nossa aventura, um *spin-off* do livro *Cangaceiro JavaScript*.

# Sumário

Versão: 22.3.31

# Parte 1 - O encontro

*Com a captura de Lampião e de seu bando, apenas um jovem cangaceiro ainda restava. Anos depois, realizando pequenos favores para sobreviver, chegou aos seus ouvidos o conto de um velho cangaceiro cego que driblara a força volante e que vivia na caatinga solitário. Na esperança de se reagrupar e encontrar mais uma vez a paz do seu espírito, pegou suas coisas e se pôs a andar. Depois de ter caminhando durante quatro horas sob o sol ardente, foi então que ele avistou um acampamento de um homem apenas. Era um velhaco que sentava sob suas pernas e que afiava vagarosamente sua peixeira. Seus olhos eram esbranquiçados e miravam o nada, confirmação de sua cegueira. Quando mais o cangaceiro se aproximava, mais forte o velho rangia a lâmina na pedra. Foi então que o cangaceiro hesitou e decidiu voltar, pois acreditava que nada de útil poderia vir daquele lugar. Ao virar suas costas, de súbito, seu chapéu foi derrubado. Acreditando ser a molecagem de algum pássaro, inclinou-se vagarosamente para pegar seu chapéu e, para sua surpresa, encontrou uma peixeira em seu centro. Antes que pudesse ter qualquer reação, o cego levantou-se e disse: "Volte, você ainda tem muito o que aprender. Ah, e traga minha peixeira!". Foi a partir desse dia que o cangaceiro voltou a ser aprendiz mais uma vez.*

Capítulo 1

# ORGANIZAÇÃO INICIAL DO PROJETO

> *"O sertanejo é, antes de tudo, um forte"*. - Os Sertões

## 1.1 SUBINDO O SERVIDOR

Com os requerimentos de infraestrutura atendidos e com o `zip` do projeto descompactado, para levantá-lo, abra seu terminal favorito e entre na pasta `cangaceiro2` . Dentro dela, execute o comando:

```
node server
```

> Se você utiliza Linux e o Node.js foi instalado como `nodejs` , troque o comando anterior para `nodejs server` .

A instrução fará com que o servidor fique de pé e exiba no console a seguinte mensagem:

```
Server is running at http://localhost:3000
```

Abra seu navegador e acesse o endereço `http://localhost:3000` . Deverá ser exibida uma página com um botão apenas:


Figura 1.1: Página com um botão apenas

Agora com tudo em seu devido lugar, podemos dar início ao projeto. Vamos começar pelo sistema de módulos do ES2015 e seu suporte nativo pelos navegadores.

## 1.2 A PÁGINA PRINCIPAL DO PROJETO

Vamos acessar o endereço `http://localhost:3000` que exibirá uma página padrão já criada que exibe um título e um botão.

Ela possui a seguinte estrutura:

```
<!-- cangaceiro2/public/index.html -->
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="css/bootstrap.css">
    <link rel="stylesheet" href="css/bootstrap-theme.css">
    <title>Notas Fiscais</title>
```

```
</head>
<body class="container">
    <h1 class="text-center">Notas Fiscais</h1>
    <div class="text-center">
        <button id="myButton" class="btn btn-primary">
            Buscar
        </button>
    </div>
</body>
</html>
```

Este arquivo fica dentro da pasta `notas/public`. Aliás, todos os arquivos estáticos do projeto ficam nessa pasta, inclusive os arquivos do Bootstrap que utilizaremos para aplicar um visual profissional em nossa aplicação com pouco esforço.

## 1.3 ORGANIZANDO O CÓDIGO EM MÓDULOS

Nossa primeira tarefa será buscar uma lista de notas fiscais. O servidor embutido no projeto que baixamos disponibiliza o endpoint REST `http://localhost:3000/notas` com esta finalidade. Porém, antes de partirmos para a implementação do nosso código, precisamos pensar primeiro como ele será organizado. Sem sombra de dúvidas, vamos organizá-lo em módulos; porém, como faremos o carregamento deles em nosso navegador?

### Suporte nativo de módulos do ES2015

A partir do ES2015 (ES6) foi introduzido um sistema de módulos padrão na linguagem caracterizado pelo uso da sintaxe `import` e `export`. Todavia, por haver divergências sobre como implementar o carregamento de módulos nos navegadores, essa

parte não foi especificada, jogando a responsabilidade do carregamento no colo de bibliotecas e pré-processadores JavaScript. Agora, a polêmica sobre o carregamento de módulos parece estar chegando ao fim.

Os principais navegadores do mercado já suportam o carregamento nativo de módulos através da tag `<script type="module">`.

Partiremos do princípio de que o leitor já conhece o sistema de módulos do ES2015, pois focaremos apenas na estratégia de seu carregamento pelo navegador, que será a primeira novidade aqui.

Caso o leitor queira saber mais sobre o sistema de módulos do ES2015 e como suportá-lo em diversos navegadores através de um processo de compilação, ele pode consultar o livro *Cangaceiro JavaScript: uma aventura no sertão da programação* (https://www.casadocodigo.com.br/products/livro-cangaceiro-javascript) deste mesmo autor.

## 1.4 O MÓDULO PRINCIPAL DA APLICAÇÃO

Vamos criar o módulo principal da aplicação `app.js` dentro da pasta `public/app`. Todos os arquivos que escreveremos ficarão dentro desta pasta. Por enquanto, `app.js` não dependerá de outros módulos, mas quando houver dependência, o navegador será inteligente para baixá-los sem termos a necessidade de importá-los através da tag `<script>`.

O comportamento descrito no parágrafo anterior é fantástico, pois remove do desenvolvedor a responsabilidade pelo carregamento manual de scripts, bastando indicar qual será o

primeiro módulo a ser carregado pela aplicação.

A primeira tarefa que faremos em `app.js` será associar um evento de clique ao único botão que existe na página. Por enquanto, exibiremos apenas um alerta para o usuário toda vez que o botão for clicado:

```
// cangaceiro2/public/app/app.js

document
.querySelector('#myButton')
.onclick = () => alert('oi');
```

Através da propriedade `onclick` do elemento com ID `myButton` e com auxílio de `arrow function`, temos um código mais terso.

## Carregando o módulo nativamente

Agora, vamos realizar o carregamento do módulo através da tag `<script type="module">`:

```
<!-- cangaceiro2/public/index.html -->
<!DOCTYPE html>
<html lang="pt-br">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link rel="stylesheet" href="css/bootstrap.css">
    <link rel="stylesheet" href="css/bootstrap-theme.css">
    <title>Notas Fiscais</title>
</head>
<body class="container">

    <h1 class="text-center">Notas Fiscais</h1>

    <div class="text-center">
```

```
        <button id="myButton" class="btn btn-primary">
            Buscar
        </button>

    </div>

    <!-- importando o módulo -->

    <script type="module" src="app/app.js"></script>
</body>
</html>
```

Por mais que tenhamos importado o módulo, ele só será carregado se estivermos acessando a aplicação através de um servidor Web e não diretamente pelo sistema de arquivo, portanto tenha a certeza de estar acessando nossa página através do endereço `http://localhost:3000` .

## 1.5 CONSUMO DE ENDPOINTS REST

Agora que já temos as coisas no lugar, vamos buscar uma lista de notas fiscais através de uma requisição Ajax com auxílio da API **Fetch**.

Vamos direto para implementação do código:

```
// cangaceiro2/public/app/app.js
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(res => res.json())
    .then(notas => console.log(notas))
    .catch(err => console.log(err));
```

Precisamos de um conhecimento mínimo da Fetch API, por isso será feita uma breve introdução ao tema.

## Introdução relâmpago à Fetch API

A Fetch API é acessível globalmente através da função `fetch` que recebe como parâmetro o endereço Web de algum recurso a ser consumido. Seu retorno é uma **Promise**, uma maneira elegante de lidarmos com o resultado futuro de uma ação.

Toda Promise possui as funções `then` e `catch`. Na primeira, obtemos o resultado da operação. Já na segunda, tratamos possíveis erros que possam ocorrer durante a resolução da nossa Promise. Porém, como podemos ver, há duas chamadas para a função `then`.

Um dos *pulos do gato* da Promise é entender que, se a função `then` retorna um valor qualquer, esse valor é acessível através de uma nova chamada **encadeada** à função `then`.

Vejamos o seguinte trecho de código:

```
// cangaceiro2/public/app/app.js

// Revisando
// código anterior omitido
fetch('http://localhost:3000/notas')
.then(res => res.json())
// código posterior omitido
```

O resultado da busca é um objeto `Response`. Podemos obter a resposta no formato texto através da função `res.text` ou seu valor já convertido para JSON através de `res.json`.

Como estamos usando uma *arrow function* sem bloco, implicitamente é adicionada a instrução `return`. Sendo assim, na próxima chamada à `then` teremos acesso aos dados retornados no formato JSON, dados que exibimos no console do navegador.

Por fim, na cláusula `catch`, passamos `console.log`. É uma forma mais enxuta do que fazermos `.catch(err => console.log(err))`.

```
// cangaceiro2/public/app/app.js

// código anterior omitido
// Revisando
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(res => res.json())
    // notas aqui é o resultado de res.json()
    .then(notas => console.log(notas))
    .catch(console.log);
    // código posterior omitido
```

Tudo muito lindo se não fosse por **uma pegadinha**. O que acontecerá se acessarmos um endereço que não existe, por exemplo, `http://localhost:3000/notasx`?

Utilizando esse endereço, veremos a seguinte mensagem de erro no console do Chrome:

```
Unexpected end of JSON input
```

Por que isso aconteceu?

Para a API Fetch, qualquer resposta vinda do servidor é uma resposta válida, inclusive a `statusMessage` do status `404` (não encontrado). Nesse sentido, estamos realizando o *parse* de uma resposta que não é um JSON! Precisamos verificar na cláusula `then` se o código do status da resposta é um código válido, para então realizarmos o *parse*.

A boa notícia é que a API Fetch nos traz o atalho `res.ok` para sabermos se a requisição é válida ou não. Ela nos poupa o trabalho

de testar cada código de status das faixas 400 e 500, por exemplo.

Vamos alterar nosso código:

```
// cangaceiro2/public/app/app.js

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notasx')
    .then(res => {
        if(res.ok) return res.json();
        return Promise.reject(res.statusText);
    })
    .then(notas => console.log(notas))
    .catch(console.log);
```

Usamos a seguinte estratégia. Se o `res.ok` é válido, retornamos o resultado de `res.json()` para a próxima chamada encadeada à `then`. Caso contrário, rejeitamos a Promise passando a mensagem de erro do status vindo do servidor. Isso fará com que a função `catch` seja acionada.

Agora, a mensagem de erro recebida no console será:

```
/notasx not found
```

Excelente, porém poderia ficar ainda melhor. Na maioria das vezes utilizaremos a mesma estratégia para lidar com o status da requisição. Sendo assim, que tal isolarmos o trecho de código que acabamos de escrever em um módulo separado? Que tal o nome `app/utils/promise-helpers.js`?

Vamos criar o novo arquivo e, nele, vamos declarar a função `handleStatus`. Aliás, vamos lançar mão de um `if` ternário para evitar o uso de um bloco em nossa *arrow function*:

```
// cangaceiro2/public/app/utils/promise-helpers.js
```

```
const handleStatus = res =>
    res.ok ? res.json() : Promise.reject(res.statusText);
```

Todavia, do jeito como declaramos, a função não poderá ser importada por nenhum outro módulo. Precisamos explicitar isso por meio da instrução `export`:

```
// cangaceiro2/public/app/utils/promise-helpers.js

// agora a função pode ser exportada por outros módulos
export const handleStatus = res =>
    res.ok ? res.json() : Promise.reject(res.statusText);
```

Agora, vamos importar a função e utilizá-la em `app/app.js`. Faremos isso através da instrução `import`. No entanto, a implementação do sistema nativo de módulos dos navegadores exige que uma extensão seja utilizada. No caso, utilizaremos `.js`. Caso a extensão seja omitida, o navegador não será capaz de carregar o módulo.

> *Frameworks como Angular ou React que utilizam Webpack por baixo dos panos não precisam adicionar a extensão, pois o módulo `bundler` se encarregará de resolvê-la.*

```
// cangaceiro2/public/app/app.js

// importou
import { handleStatus } from './utils/promise-helpers.js';

// ainda estamos usando o endereço errado
document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notasx')
```

```
.then(handleStatus)
.then(notas => console.log(notas))
.catch(console.log);
```

Muito mais legível, não? Não se esqueça de realizar alguns testes depois e corrigir o endereço utilizado para que possamos continuar.

Se olharmos a aba *network* do Chrome, constataremos que o módulo `promise-helpers.js` foi carregado inteligentemente pelo navegador que detectou a dependência no módulo principal da aplicação `app.js`.

Agora que já temos a espinha dorsal do nosso projeto pronta, podemos avançar em nossa aplicação.

> Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part1/01

CAPÍTULO 2

# ADEQUAÇÃO DOS DADOS RECEBIDOS

> *"Sertão é onde o pensamento da gente se forma mais forte do que o lugar."* - Grande Sertão: Veredas

Somos capazes de buscar uma lista de notas fiscais, excelente. Todavia, precisamos totalizar o valor dos itens com o código `2143` de todas as notas.

Vamos acessar o recurso `http://localhost:3000/notas` por meio do navegador. Receberemos um JSON com esta estrutura:

```
[
    {
        data: '2017-10-31',
        itens: [
            { codigo: '2143', valor: 200 },
            { codigo: '2111', valor: 500 }
        ]
    },
    {
        data: '2017-07-12',
        itens: [
```

```
            { codigo: '2222', valor: 120 },
            { codigo: '2143', valor: 280 }
        ]
    },
    {
        data: '2017-02-02',
        itens: [
            { codigo: '2143', valor: 110 },
            { codigo: '7777', valor: 390 }
        ]
    },
];
```

Olhando a estrutura de dados retornada e realizando o cálculo mentalmente, o valor total de todos os itens com código `2143` será R$ 590,00. Mas apenas realizar o cálculo mentalmente não vale, cangaceiros são pessoas de ação e por isso vamos implementá-lo programaticamente!

Antes de partirmos para a implementação do código, façamos um *checklist* do que precisamos fazer:

1. obter uma lista com todos os itens de todas as notas fiscais;
2. filtrar todos os itens pelo código `2143` ;
3. somar o valor de todos esses itens.

Agora que já temos uma ideia do que fazer, vamos dar início à nossa implementação.

## 2.1 ARRAY E SEUS SUPERPODERES

Em JavaScript, podemos realizar cada etapa descrita na seção anterior por meio das funções:

1. `Array.map`
2. `Array.filter`

3. `Array.reduce`

Agora podemos partir para o código:

```
// cangaceiro2/public/app/app.js

import { handleStatus } from './utils/promise-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)

    // retornará para o próximo then uma lista de itens
    .then(notas => notas.map(nota => nota.itens))

    // retornará para o próximo then uma lista de itens filtrada
    .then(itens => itens.filter(item => item.codigo == '2143'))

    // retornará para o próximo then o total
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))

    // exibe o resultado
    .then(total => console.log(total))
    .catch(console.log);
```

Como só temos interesse em exibir o total no console, podemos simplificar a chamada da mesma maneira que fizemos com o erro capturado em `catch`:

```
// cangaceiro2/public/app/app.js

import { handleStatus } from './utils/promise-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.map(nota => nota.itens))
    .then(itens => itens.filter(item => item.codigo == '2143'))
```

```
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    // simplificando
    .then(console.log)
    .catch(console.log);
```

Apesar dos nossos esforços, o resultado será `0`. A causa desse resultado não esperado está no mapeamento do array de notas fiscais para um array de itens por meio da instrução `.then(notas => notas.map(nota => nota.itens))`. O mapeamento realizado gerou um array multidimensional, algo que podemos conferir com nossos próprios olhos exibindo no console o resultado do mapeamento através de uma chamada extra à função `then`:

```
// cangaceiro2/public/app/app.js

import { handleStatus } from './utils/promise-helpers.js';

document
.querySelector('#myButton')
    .onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.map(nota => nota.itens))
    // then extra!
    .then(itens => {
        console.log(itens);
        return itens;
    })
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    .then(console.log)
    .catch(console.log);
```

Veremos como saída do console:

```
[Array(2), Array(2), Array(2)]
```

Não temos um array de única dimensão com todos os itens,

temos um array multidimensional, sendo que cada array corresponde à lista de itens de cada nota. E agora?

Precisamos reduzir todos os itens do array para uma dimensão apenas. Podemos fazer isso trocando `map` por `reduce`, aliás, é uma boa hora para revisarmos como essa poderosa função funciona. Para isso, vamos usar um exemplo de escopo menor.

## 2.2 REVISANDO A FUNÇÃO REDUCE

Precisamos calcular a média aritmética das idades de uma lista. Sabemos que para tal precisamos somar todos os itens da lista para, em seguida, dividir o total calculado pela quantidade de itens.

Uma implementação possível é a seguinte:

```
// exemplo apenas, não entra na aplicação
const idades = [26, 18, 42];
let total = 0;
for(let idade of idades) {
    total+=idade;
}
const media = total/idades.length;
console.log(media);
```

Uma solução totalmente válida. Porém, através de `reduce` podemos conseguir o mesmo resultado sem termos que assumir a responsabilidade pela iteração e acúmulo dos valores:

```
// exemplo apenas, não entra na aplicação
const idades = [26, 18, 42];
const total = idades.reduce((total, idade) => total+=idade, 0);
const media = total/idades.length;
console.log(media);
```

A função `reduce` recebe dois parâmetros. O primeiro é a função que será aplicada em cada item da lista. Ela recebe como

primeiro parâmetro a variável que acumulará o resultado a cada iteração e, como segundo, o elemento que está sendo iterado no momento. É por este motivo que em seu bloco temos a instrução `total+=idade` , garantindo que `total` será acrescido do valor da `idade` iterada. O segundo parâmetro de `reduce` é o valor inicial que a variável acumuladora terá na primeira iteração. Em nosso caso, o valor inicial de total será zero.

Por fim, podemos omitir o valor de inicialização de `total` . Quando o valor é omitido, o `reduce` considerará o primeiro item do array como valor inicial de `total` , isto é, nossa variável acumuladora:

```
// exemplo apenas, não entra na aplicação
const idades = [26, 18, 42];
const total = idades.reduce((total, idade) => total+=idade);
const media = total/idades.length;
console.log(media);
```

Podemos enxugar ainda mais nosso código evitando a declaração da variável `total` :

```
// exemplo apenas, não entra na aplicação
const idades = [26, 18, 42];
const media = idades.reduce((total, idade) =>
    total+=idade) / idades.length;
console.log(media);
```

Agora que já revisamos como `reduce` funciona, podemos dar início à solução do nosso problema com sua ajuda.

## 2.3 ARRAYS DE UMA DIMENSÃO ATRAVÉS DE REDUCE

Sabemos que `reduce` recebe um callback como primeiro

parâmetro e um valor de inicialização para a variável acumuladora disponível como primeiro parâmetro no callback. Com base nesse conceito, podemos reduzir a lista para uma dimensão apenas com o seguinte trecho de código:

```
// exemplo, ainda não entra no código
notas.reduce((array, nota) => array.concat(nota.itens), []))
```

A variável acumuladora `array` será inicializada com `[]`, um array vazio. A cada iteração adicionaremos os elementos de `nota.itens` em `array` por meio de `array.concat(nota.itens)`. A função `concat` é inteligente o suficiente para receber uma lista de itens e adicionar cada item diretamente no array que chamou a operação de concatenação.

Transpondo a solução que acabamos de construir para nosso código, ele fica assim:

```
// cangaceiro2/public/app/app.js

import { handleStatus } from './utils/promise-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.reduce((array, nota) => array.concat(not
a.itens), []))
    .then(itens => {
        // temos agora um array de única dimensão
        console.log(itens);
        return itens;
    })
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    .then(console.log)
    .catch(console.log);
```

Conseguimos chegar ao resultado R$ 590,00. Podemos até visualizar a lista filtrada antes de ela chegar ao `reduce` :

```
// cangaceiro2/public/app/app.js

import { handleStatus } from './utils/promise-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.reduce((array, nota) => array.concat(not
a.itens), []))
    .then(itens => {
        console.log(itens);
        return itens;
    })
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(itens => {
        // nova lista filtrada
        console.log(itens);
        return itens;
    })
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    .then(console.log)
    .catch(console.log);
```

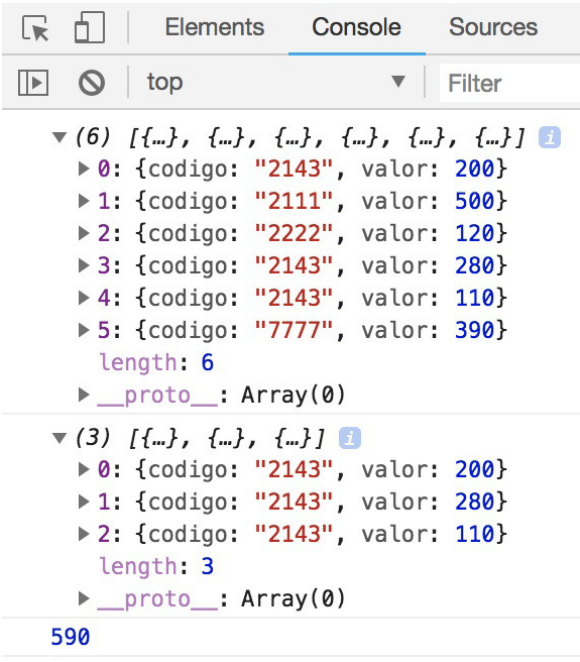



Figura 2.1: Saída no console

Tudo funcionando, porém um olhar atento reparará que repetimos a seguinte instrução em mais de um lugar:

```
// cangaceiro2/public/app/app.js
// código anterior omitido
// instrução repetida

.then(itens => {
    console.log(itens);
    return itens;
})

// código posterior omitido
```

Podemos tornar nosso código ainda melhor isolando a responsabilidade pelo log em uma função. Aliás, este é o assunto

da próxima seção.

## 2.4 IMPLEMENTAÇÃO DA FUNÇÃO LOG

Vamos criar a função `log`, aquela que será a responsável pela exibição no console do dado retornado pela função `then`. Inclusive ela será declarada no módulo `promise-helpers.js` devido a sua forte relação com Promise:

```
// cangaceiro2/public/app/utils/promise-helpers.js

export const handleStatus = res =>
    res.ok ? res.json() : Promise.reject(res.statusText);

export const log = param => {
    console.log(param);
    return param;
};
```

Agora, em `app.js`, vamos importar a função e utilizá-la:

```
// cangaceiro2/public/app/app.js

import { handleStatus, log } from './utils/promise-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.reduce((array, nota) => array.concat(not
a.itens), []))
    .then(log) // chamou log
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(log) // chamou log
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    .then(log) // chamou log
    .catch(log); // chamou log
```

Excelente! Temos menos código duplicado, mas ainda podemos organizá-lo melhor, no que diz respeito à sua intenção.

## 2.5 A INTENÇÃO DO CÓDIGO

Vamos voltar nossos olhos para a seguinte chamada:

```
// cangaceiro2/public/app/app.js
// código anterior omitido

.then(notas => notas.reduce((array, nota) => array.concat(nota.it
ens), []))

// código posterior omitido
```

Apesar de funcionar, essa chamada não deixa clara nossa intenção, que é a de realizarmos um `map` cujo resultado tenha uma dimensão apenas.

A boa notícia é que está em andamento uma proposta para a linguagem de um `map` especial chamado `flatMap` que retorna os dados em uma dimensão apenas, justamente o que precisamos. A má notícia? Na data de publicação deste livro, a função `flatMap` ainda não foi implementada nos navegadores. O que fazer?

Uma solução é nós mesmos implementarmos a função `flatMap` para então adicioná-la como função válida de um array. Isso é possível devido à característica dinâmica da linguagem JavaScript, que permite modificar estruturas já existentes.

## 2.6 IMPLEMENTAÇÃO DA FUNÇÃO FLATMAP

Vamos implementar a função `flatMap` , mas primeiro vamos

comparar o trecho de código que temos hoje com o que pretendemos ter:

Atualmente, temos o seguinte código:

```
// atual

notas.reduce((array, nota) => array.concat(nota.itens), [])
```

Com `flatMap` ele ficará assim:

```
// como ficará

notas.$flatMap(notas => notas.itens)
```

Adicionaremos a função `$flatMap` no `prototype` de `Array`, permitindo que todos os arrays criados em nossa aplicação tenham acesso à função. Vamos criá-la no módulo `cangaceiro2/public/app/utils/array-helpers.js`:

```
// cangaceiro2/public/app/utils/array-helpers.js
// só adicionada no prototype se não existir

if(!Array.prototype.$flatMap) {
    Array.prototype.$flatMap = function(cb) {
        return this.map(cb).reduce((destArray, array) =>
            destArray.concat(array), []);
    }
}
```

Tivemos que usar uma `function` no lugar de uma `arrow function` pois precisamos que o `this` seja dinâmico, referenciando a instância do array que está executando a função em dado momento.

A função `$flatMap` recebe como parâmetro a lógica do `map` que será empregada. Em nosso caso, passaremos a lógica `notas => notas.itens` porque queremos um array multidimensional de

itens.

Por fim, o `reduce` garantirá que o array terá uma dimensão apenas concatenando cada item do array em um novo array.

Um ponto curioso é que nosso módulo não exportará nada. Mas como assim? Queremos apenas que nosso módulo seja carregado para que a operação de adição da função `$flatMap` seja executada.

Vamos voltar para `cangaceiro2/public/app/app.js` e importar o módulo da seguinte maneira:

```
// cangaceiro2/public/app/app.js

import { handleStatus, log } from './utils/promise-helpers.js';
// novo import!
import './utils/array-helpers.js';
```

Repare que apenas importamos o módulo passando seu caminho.

Agora podemos modificar nosso código, que ficará assim:

```
// cangaceiro2/public/app/app.js
import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.$flatMap(nota => nota.itens))
    .then(log)
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(log)
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    .then(log)
```

```
    .catch(log);
```

Como já temos uma visão geral do código que escrevemos, não são mais necessárias as chamadas `.then(log)`. Vamos removê-las:

```
// cangaceiro2/public/app/app.js

import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas.$flatMap(nota => nota.itens))
    .then(itens => itens.filter(item => item.codigo == '2143'))
    .then(itens => itens.reduce((total, item) => total + item.val
or, 0))
    .then(log)
    .catch(log);
```

> Agora que já temos nossa implementação de `flatMap` disponível em qualquer array da nossa aplicação, vale destacar que a estrutura `Array` em JavaScript é um **Functor**. No jargão da Programação Funcional, um *Functor* é simplesmente uma estrutura que encapsula um valor e que permite realizar a operação `map` com o valor encapsulado.

Ótimo! Nosso código continua com o mesmo comportamento de antes, pois só mudamos sua estrutura. Aliás, tratando-se de estrutura, podemos realizar mais uma modificação para escrevermos ainda menos.

Como não estamos mais verificando a passagem de dados de um `then` para outro, podemos reescrever nosso código da seguinte maneira:

```
// cangaceiro2/public/app/app.js

import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(notas => notas
        .$flatMap(nota => nota.itens)
        .filter(item => item.codigo == '2143')
        .reduce((total, item) => total + item.valor, 0))
    .then(log)
    .catch(log);
```

Ainda mais enxuto, não? Todavia nosso código ainda deixa a desejar e veremos a razão disso na próxima seção.

## 2.7 DEIXANDO CLARA NOSSA INTENÇÃO

Vamos olhar o seguinte trecho do nosso código:

```
// cangaceiro2/public/app/app.js
// apenas verificando o código existente
// código anterior omitido

.then(notas => notas
    .$flatMap(nota => nota.itens)
    .filter(item => item.codigo == '2143')
    .reduce((total, item) => total + item.valor, 0))

// código posterior omitido
```

A intenção do trecho de código que acabamos de ver é somar

todos os itens da nota fiscal. Que tal se o mesmo código fosse escrito da seguinte maneira:

```
// exemplo

.then(sumItems)
```

Muito mais claro e rápido de entender, não? Aliás, não precisamos ficar apenas na vontade, podemos criar imediatamente a função `sumItems`:

```
// cangaceiro2/public/app/app.js

import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

const sumItems = notas => notas
    .$flatMap(nota => nota.itens)
    .filter(item => item.codigo == '2143')
    .reduce((total, item) => total + item.valor, 0);

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(sumItems)
    .then(log)
    .catch(log);
```

Muito bom, mas e se agora quisermos somar todos os itens de outra conta? Nossa função `sumItems` não está preparada para isso, mas podemos adequá-la com um pequeno esforço.

Vamos alterar `sumItems` para que receba como parâmetro um código e retorne **uma nova função** que lembrará do código recebido. A nova função retornada receberá um array de notas fiscais e será ela que passaremos para `then`. A função retornada consegue lembrar do parâmetro da função que a retornou devido

ao suporte que a linguagem JavaScript dá a *closure*, que consiste na lembrança do ambiente no qual a função foi declarada. Vamos recordar esse poderoso recurso da linguagem com um exemplo de escopo menor.

## 2.8 REVISANDO CLOSURE

Temos a função `funcaoExterna` que recebe o parâmetro `nome` e que, ao ser chamada, retorna outra função:

```
// exemplo apenas
const funcaoExterna = nome => {
    const hoje = new Date();
    return () => console.log(nome, hoje);
};
```

Para fins didáticos, segue o código anterior em sua forma mais verbosa sem o uso de *arrow functions*:

```
// exemplo apenas
const funcaoExterna = function(nome) {
    const hoje = new Date();
    return function () {
        console.log(nome, hoje);
    };
};
```

Executando a função:

```
const funcaoInternaRetornada = funcaoExterna('Flávio');
funcaoInternaRetornada();
// exibe no console:
// Flávio Tue Jun 19 2018 12:44:44 GMT-0300 (Horário Padrão de Br
asília)
```

A função interna retornada por `funcaoExterna` não apenas lembra do parâmetro recebido pela função mais externa, mas também da variável `hoje` declarada no contexto da mesma

função, caracterizando o suporte a *closure*. Trata-se da capacidade de uma função retornada lembrar do contexto no qual foi declarada. Esse contexto traz consigo as variáveis e outras funções que tenham sido declaradas na função externa.

Agora que já revisamos essa característica fantástica da linguagem JavaScript, já podemos tirar da algibeira nossa implementação cangaceira!

## 2.9 MODIFICANDO FUNÇÕES PARA QUE RECEBAM APENAS UM PARÂMETRO

Vamos alterar a função `sumItems` para que receba um parâmetro apenas:

```
// cangaceiro2/public/app/app.js

import { handleStatus, log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';

const sumItems = code => notas => notas
    .$flatMap(nota => nota.itens)
    .filter(item => item.codigo == code)
    .reduce((total, item) => total + item.valor, 0);

document
.querySelector('#myButton')
.onclick = () =>
    fetch('http://localhost:3000/notas')
    .then(handleStatus)
    .then(sumItems('2143')) // sumItems retorna nova função
    .then(log)
    .catch(log);
```

Uma pequena alteração com grande impacto na legibilidade. Será que podemos fazer mais pelo nosso código? Sim, podemos, e é isso que nós veremos a seguir.

## 2.10 ISOLAMENTO DE CÓDIGO POR MEIO DE UM SERVIÇO

Vamos dar uma olhadinha na seguinte chamada da função `fetch`:

```
fetch('http://localhost:3000/notas')
```

Teremos esse código espalhado em todos os locais da nossa aplicação que precisarem buscar negociações e, se o endereço mudar, teremos que alterar em todos os lugares. Além disso, esse trecho de código por si só não deixa clara sua intenção.

Podemos isolar o acesso à API em uma classe de serviço. Em todos os lugares em que precisamos interagir com a API, vamos fazê-lo por meio desta classe.

Vamos criar a pasta e o módulo `cangaceiro2/public/app/nota/service.js`. Quando ele for importado, devolverá sempre um objeto que representa o serviço. Aliás, vamos isolar a chamada de `handleStatus` e tratar qualquer erro de baixo nível que aconteça lançando em seu lugar uma mensagem de alto nível:

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';

const API = 'http://localhost:3000/notas';

export const notasService = {
    listAll() {
        return fetch(API)
        // lida com o status da requisição
        .then(handleStatus)
        .catch(err => {
            // o serviço agora é o responsável em logar o erro
```

```
            console.log(err);
            // retorna uma mensagem amigável
            return Promise.reject('Não foi possível obter as nota
s fiscais');
        });
    }
};
```

Não há necessidade de utilizarmos uma classe aqui porque não queremos criar instâncias de uma classe. Nosso serviço não guardará estado, a não ser a URL da API que ficará no escopo do módulo e, por meio de *closure*, o objeto retornado terá acesso à variável `API`.

Agora, vamos utilizá-lo em `cangaceiro2/public/app/app.js`:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
// importa dando um apelido para o artefato importado
import { notasService as service } from './nota/service.js';

const sumItems = code => notas => notas
    .$flatMap(nota => nota.itens)
    .filter(item => item.codigo == code)
    .reduce((total, item) => total + item.valor, 0);

document
.querySelector('#myButton')
.onclick = () =>
    // utilizando o serviço
    service
    .listAll()
    .then(sumItems('2143'))
    .then(log)
    .catch(log);
```

Uma coisa diferente que realizamos foi a importação de `notasService` e seu uso por meio de um apelido que permitiu

encurtar o nome do artefato importado. Porém, esse recurso é interessante quando diferentes módulos exportam artefatos de mesmo nome e no momento da importação somos obrigados a lançar mão desse artifício para resolver a ambiguidade.

Nosso código começa a ganhar forma, mas há mais uma melhoria que se torna oportuna. Se precisarmos obter o total de todos os itens a partir de um código em outros lugares da nossa aplicação, repetiremos código. Nesse sentido, vamos isolar a lógica de `sumItems` no próprio serviço a partir de um novo método que utilizará o já existente `listAll()`.

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';

const API = 'http://localhost:3000/notas';

const sumItems = code => notas => notas
    .$flatMap(nota => nota.itens)
    .filter(item => item.codigo == code)
    .reduce((total, item) => total + item.valor, 0);

export const notasService = {

    listAll() {
        return fetch(API)
        .then(handleStatus)
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as nota
s fiscais');
        });
    },
    // novo método
    sumItems(code) {
        return this.listAll().then(sumItems(code));
    }
};
```

Por fim, nosso `app.js` ficará assim:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';

document
.querySelector('#myButton')
.onclick = () =>
    service
    .sumItems('2143')
    .then(log)
    .catch(log);
```

Excelente! Terminamos este capítulo com um código mais bem organizado. Mas será que podemos torná-lo ainda melhor? É isso que veremos no próximo capítulo.

> Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part1/02

Capítulo 3

# COMPOSIÇÃO DE FUNÇÕES

> *"Não conto anedota porque não convém, a alegria que eu tenho não dou a ninguém."* - Pão Duro, Luiz Gonzaga.

Vamos voltar nossa atenção para a função `sumItems`:

```
// cangaceiro2/public/app/nota/service.js

// código anterior omitido

const sumItems = code => notas => notas
    .$flatMap(nota => nota.itens)
    .filter(item => item.codigo == code)
    .reduce((total, item) => total + item.valor, 0);

// código posterior omitido
```

Não precisamos meditar muito para chegarmos à conclusão de que a função faz muita coisa. Ela:

1. obtém uma lista de itens de uma lista de notas fiscais;
2. filtra a lista de itens por um código;
3. totaliza o valor dos itens.

Para que nosso código fique ainda mais claro e mais fácil de manter, vamos extrair cada responsabilidade que acabamos de listar em sua respectiva função:

```
// cangaceiro2/public/app/app.js

// código anterior omitido

// não existe mais a função `sumItemsWithCode`. Ela foi substituí
da por três funções

const getItemsFromNotas = notas => notas
    .$flatMap(nota => nota.itens);

const filterItemsByCode = (code, items) => items
    .filter(item => item.codigo === code);

const sumItemsValue = items => items
    .reduce((total, item) => total + item.valor, 0);

// código posterior omitido
```

Agora que temos três funções com responsabilidades bem definidas, precisamos combiná-las para criarmos a extinta função `sumItem`. Em outras palavras, queremos realizar a **composição de funções**.

Na matemática, composição de funções consiste na aplicação de uma função ao resultado de outra função, sucessivamente. Todavia, nossas funções precisam receber apenas um parâmetro e, para atrapalhar um pouco nossa tarefa, a função `filterItemsByCode` recebe duas. E agora?

## 3.1 REDUZINDO A QUANTIDADE DE PARÂMETROS DE UMA FUNÇÃO

Podemos fazer com que a função `filterItemsByCode` receba

um parâmetro apenas, no caso, o código que será utilizado durante o filtro dos itens. Seu retorno será uma função que receberá a lista de notas fiscais que precisa operar. Esta última função ainda que seja retornada lembrará do parâmetro do código recebido pela primeira função através de *closure*:

```
// apenas exemplo, não entra em nosso código

const getItemsFromNotas = notas => notas
    .$flatMap(nota => nota.itens);

// alterando a função
const filterItemsByCode = code => items => items
    .filter(item => item.codigo === code);

const sumItemsValue = items => items
    .reduce((total, item) => total + item.valor, 0);
```

Simulando o uso da função, ela funcionará dessa forma:

```
// apenas exemplo, não entra em nosso código

const filterItems = filterItemsByCode('2143');
// função pronta para filtrar itens
```

Excelente, mas não utilizaremos essa solução. O motivo? Se tivéssemos outros lugares da nossa aplicação que fizessem uso de `filterItemsByCode`, eles quebrariam!

A ideia é criarmos uma nova função baseada em `filterItemsByCode` que já aplique **parcialmente** o primeiro parâmetro da função, no caso, o código. Com essa abordagem, a função receberá apenas um parâmetro, no caso, a lista de notas fiscais. No fim, o que queremos é realizar **partial application** com a função `filterItemsByCode`.

## 3.2 PARTIAL APPLICATION

Para que possamos compreender mais facilmente a *partial application*, inclusive sua implementação, trabalharemos com um problema de escopo menor para então transpormos a solução para o problema que estamos tentando resolver.

Temos a função `ehDivisivel`, que recebe dois parâmetros. O primeiro é o divisor e o segundo, o número que será testado. Verificaremos se três números são divisíveis por dois:

```
// apenas exemplo, não entra na aplicação

const ehDivisivel = (divisor, numero) => !(numero % divisor);
ehDivisivel(2, 10); // true
ehDivisivel(2, 15); // false
ehDivisivel(2, 20); // true
```

Todavia, com *partial application* podemos memorizar o divisor evitando ter de passá-lo como argumento para a função toda vez que for utilizada. Conseguimos isso facilmente através de um recurso da própria linguagem JavaScript.

## Partial application com auxílio da função bind

Podemos realizar a *partial application* de funções por meio da função `Function.bind()`. Vejamos a função em ação:

```
// apenas exemplo, não entra na aplicação

const ehDivisivel = (divisor, numero) => !(numero % divisor);

// criou uma função parcial
const ehDivisivelPorDois = ehDivisivel.bind(null, 2);

ehDivisivelPorDois(10); // true
ehDivisivelPorDois(15); // false
ehDivisivelPorDois(20); // true
```

A função `Function.bind` cria uma nova função. Seu

primeiro argumento é o valor de `this` que desejamos que a nova função utilize como contexto. Porém, como declaramos a função por meio de *arrow function*, que não aceita a modificação do seu contexto, simplesmente passamos `null`. Mesmo que tivéssemos passado outro valor, ele seria ignorado.

Os demais parâmetros são todos aqueles que desejamos assumir como argumentos já fornecidos toda vez que a função resultante de `bind()` for chamada. No caso, estamos indicando que o primeiro parâmetro será sempre `2`.

Podemos passar quantos parâmetros desejarmos. Vejamos outro exemplo:

```
// exemplo apenas, não entra na aplicação

const ehDivisivel = (divisor, numero) => !(numero % divisor);

// assume como parâmetros 2 e 5, nesta ordem
const fn = ehDivisivel.bind(null, 2, 5);

fn(); // false
```

Agora já podemos aplicar a solução que vimos com a função `filterItemsByCode`. Todavia, essa aplicação deve ser feita dentro do método `sumItems` do nosso serviço, porque é nele que temos acesso ao código:

```
// cangaceiro2/public/app/nota/service.js
// código anterior omitido

// função continua com a estrutura original
const filterItemsByCode = (code, items) => items.filter(item => i
tem.codigo
=== code);

// código anterior omitido
```

```
sumItems(code) {

    // criando uma função parcial que ainda vamos utilizar
    const filterItems = filterItemsByCode.bind(null, code);

    // ainda falta realizar a composição dessa função com as
    // outras para criar novamente sumItems, que não existe
    return this.listAll().then(sumItems(code));
}
// código posterior omitido
```

Excelente, mas pode ficar ainda melhor!

## A função partialize

Que tal isolarmos a lógica que realiza a aplicação parcial em uma função utilitária para que possamos usar sempre que necessário? Vamos criar a função **partialize** no **novo** módulo `cangaceiro2/public/app/utils/operators.js`:

```
// cangaceiro2/public/app/utils/operators.js

export const partialize = (fn, ...params) =>
    fn.bind(null, ...params);
```

A função `partialize` recebe dois parâmetros. O primeiro é a função que desejamos reduzir a um parâmetro apenas, já o segundo, uma quantidade indeterminada dos parâmetros que desejamos que a função `fn` assuma. Conseguimos essa flexibilidade na quantidade de parâmetros aplicando o *Rest Operator* (os famosos três pontinhos) antes do parâmetro `params`. Isso fará com que `params` seja um array recebendo o segundo e próximos parâmetros.

Por fim, a função realiza o `bind` de `fn` com os parâmetros presentes no array `params` por meio de `fn.bind(null, ...params)`. Dessa vez, os três pontinhos servem para considerar

cada item do array como parâmetro para a função `bind`, recurso chamado *Spread Operator*. Em suma, se `params` contiver dez elementos, a função `fn.bind` receberá 11 parâmetros.

Agora só nos resta incorporar a nova função em `app/nota/service.js`, que ainda continuará incompleto:

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';
import { partialize } from '../utils/operators.js';

const API = 'http://localhost:3000/notas';

const getItemsFromNotas = notas =>
    notas.$flatMap(nota => nota.itens);

const filterItemsByCode = (code, items) =>
    items.filter(item => item.codigo === code);

const sumItemsValue = items =>
    items.reduce((total, item) => total + item.valor, 0);

export const notasService = {

    listAll() {
        return fetch(API)
        .then(handleStatus)
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as nota
s fiscais');
        });
    },

    sumItems(code) {

        // utilizando partialize!
        const filterItems = partialize(filterItemsByCode, code);

        // ainda falta realizar a composição dessa função com as
        // outras para criar novamente sumItems, que não existe
```

```
        return this.listAll().then(sumItems(code));
    }
};
```

Agora que já resolvemos a questão dos parâmetros da função `filterItemsByCode`, podemos partir para realização da composição das funções que criamos.

## 3.3 COMPONDO FUNÇÕES

Vamos alterar o módulo `cangaceiro2/public/app/nota/service.js`. Nele, faremos uma composição envolvendo as funções `sumItemsValue`, `filterItem` e `getItemsFromNotas`. O resultado da aplicação de todas essas funções será passado para a função `then` da Promise:

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';
import { partialize } from '../utils/operators.js';

const API = 'http://localhost:3000/notas';

const getItemsFromNotas = notas =>
    notas.$flatMap(nota => nota.itens);

const filterItemsByCode = (code, items) =>
    items.filter(item => item.codigo === code);

const sumItemsValue = items =>
    items.reduce((total, item) => total + item.valor, 0);

export const notasService = {

    listAll() {
        return fetch(API)
        .then(handleStatus)
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as nota
```

```
s fiscais');
        });
    },

    sumItems(code) {

        // utilizando partialize
        const filterItems = partialize(
            filterItemsByCode,
            code
        );

        // realizando a composição
        return this
            .listAll()
            .then(notas =>
                sumItemsValue(
                    filterItems(
                        getItemsFromNotas(notas)
                    )
                )
            );
    }
};
```

Melhoramos nosso código separando a responsabilidade da função `sumItem` em mais de uma função, mas realizar manualmente a composição deixa um tanto a desejar. Será que podemos criar uma função utilitária no estilo de `partialize`, mas que nos ajude na composição de funções? Sim, podemos, e é exatamente isso que veremos a seguir.

## Esboçando uma solução mais elegante

Podemos simplificar a composição de funções criando uma função utilitária que se encarregará de compor as funções para nós. Antes de partirmos para sua implementação, vamos vislumbrar seu uso em nosso código:

```
// cangaceiro2/public/app/nota/service.js

// exemplo de uso, não entra ainda em nossa aplicação

sumItems(code) {
    const filterItems = partialize(
        filterItemsByCode,
        code
    );

    // função compose ainda não foi implementada
    const sumItems = compose(
        sumItemsValue,
        filterItems,
        getItemsFromNotas
    );

    return this.listAll().then(sumItems);
}
```

Que tal? Muito mais enxuto e legível do que a versão anterior. Mas há mais um detalhe implícito no código que escrevemos que precisa ser desvelado a seguir.

## Point-free style

Vamos comparar a maneira pela qual realizávamos a composição de função com a nova forma.

A forma anterior:

```
// apenas para relembrar, não entra na aplicação
// código anterior omitido

sumItems(code) {
    return this.listAll().then(notas =>
        sumItemsValue(
            filterItems(
                // há referência para o parâmetro notas
                getItemsFromNotas(notas)
            )
```

```
        )
    );
}

// código posterior omitido
```

A forma atual:

```
sumItems(code) {
    const filterItems = partialize(
        filterItemsByCode,
        code
    );

    // função compose ainda não foi implementada
    const sumItems = compose(
        sumItemsValue,
        filterItems,
        getItemsFromNotas
    );

    // em nenhum momento precisamos
    // referenciar um parâmetro
    return this.listAll().then(sumItems);
}
```

Um olhar atento também reparará que não precisamos, durante a composição, referenciar qualquer parâmetro, diferente da abordagem anterior na qual o parâmetro `notas` era referenciado. Essa estética do nosso código possui um nome: **point-free style**.

Agora que já sabemos aonde queremos chegar, vamos implementar a função `compose`.

## 3.4 IMPLEMENTANDO A FUNÇÃO COMPOSE

Vamos declarar a função `compose` no novo módulo `cangaceiro2/public/app/utils/operators.js`. A função

receberá como parâmetro um número indeterminado de funções e retornará uma função que terá apenas um argumento. No contexto do problema que estamos resolvendo, essa função representará a extinta função `sumItemsWithCode`.

Quando a função retornada por `compose` for chamada, o argumento recebido será passado para a última função passada para `compose`. O resultado desta função será passado para a função anterior e assim sucessivamente, até que tenhamos um único resultado, que é o resultado da aplicação de todas as funções recebidas por `compose`.

Vamos ao código da função `compose`:

```
// cangaceiro2/public/app/utils/operators.js

// código anterior omitido
export const compose = (...fns) => value =>
    fns.reduceRight((previousValue, fn) =>
        fn(previousValue), value);
```

Como queremos reduzir a lista de funções a um valor apenas no final, nada mais justo do que usarmos a função `reduce`. No entanto, como queremos aplicar as funções da direita para a esquerda, usamos `reduceRight`.

Vimos que, ao usarmos `reduce`, podemos indicar qual será o valor inicial utilizado na redução, isso também é possível com `reduceRight`. Em nosso caso, usamos `value` que será o array recebido por `sumItemsWithCode`. Esse valor será o `previousValue` na primeira chamada de `fn(previousValue)`.

Em nosso contexto, na primeira iteração de `reduceRight`, `fn` será a função `getItemsFromNotas`. Seu resultado será o `previousValue` passado para a função anterior e assim

sucessivamente.

Agora que temos nossa função pronta, vamos importá-la e utilizá-la em `cangaceiro2/public/app/app.js` :

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';
// importou compose
import { partialize, compose } from '../utils/operators.js';

const API = 'http://localhost:3000/notas';

const getItemsFromNotas = notas =>
    notas.$flatMap(nota => nota.itens);

const filterItemsByCode = (code, items) =>
    items.filter(item => item.codigo === code);

const sumItemsValue = items =>
    items.reduce((total, item) => total + item.valor, 0);

export const notasService = {

    listAll() {
        return fetch(API)
        .then(handleStatus)
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as nota
s fiscais');
        });
    },

    sumItems(code) {

        // realizando a composição
        const sumItems = compose(
            sumItemsValue,
            partialize(filterItemsByCode, code),
            getItemsFromNotas
        );
```

```
        return this.listAll().then(sumItems);
    }
};
```

Ótimo! Alteramos a estrutura do nosso código em prol da legibilidade e manutenção sem alterar seu comportamento. Podemos fazer mais por ele, assunto da próxima seção.

## 3.5 COMPOSE VERSUS PIPE

Na seção anterior, criamos a função `compose` para nos ajudar na composição de funções. Ela se coaduna com a definição de composição da matemática, no entanto, para nós, programadores, a ordem das funções passadas para `compose` é atípica.

Vamos recapitular esse trecho de código:

```
// código extraído para flexão

const sumItems = compose(
    sumItemsValue,
    partialize(filterItemsByCode, code),
    getItemsFromNotas
);
```

As funções serão aplicadas da direita para a esquerda, no entanto, pode fazer mais sentido lermos as funções da direita para a esquerda:

```
// não terá o resultado esperado!

const sumItems = compose(
    getItemsFromNotas
    partialize(filterItemsByCode, '2143'),
    sumItemsValue,
);
```

Apesar de o código anterior não retornar o resultado esperado,

a ordem dos parâmetros deixa clara a ordem na qual as funções serão aplicadas.

Em sistemas operacionais baseados no Unix, há o operador `|` (pipe) que permite passar a saída de uma operação para a outra e assim sucessivamente.

A abordagem que acabamos de ver parece ser mais natural para nós, e podemos consegui-la criando uma nova função utilitária, que chamaremos de `pipe`.

A função `pipe` é praticamente idêntica à função `compose`, com a diferença de que as funções serão aplicadas da esquerda para a direita por meio da função `reduce` e não mais `reduceRight`:

```
// cangaceiro2/public/app/utils/operators.js

// código anterior omitido

// nova função!
export const pipe = (...fns) => value =>
  fns.reduce((previousValue, fn) =>
      fn(previousValue), value);
```

Agora, vamos alterar o módulo `cangaceiro2/public/app/app.js` para fazer uso da função `pipe` no lugar de `compose`:

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';
// importou pipe no lugar de compose
import { partialize, pipe } from '../utils/operators.js';

const API = 'http://localhost:3000/notas';

const getItemsFromNotas = notas =>
    notas.$flatMap(nota => nota.itens);
```

```
const filterItemsByCode = (code, items) =>
    items.filter(item => item.codigo === code);

const sumItemsValue = items =>
    items.reduce((total, item) => total + item.valor, 0);

export const notasService = {

    listAll() {
        return fetch(API)
        .then(handleStatus)
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as nota
s fiscais');
        });
    },

    sumItems(code) {
        // usando pipe e alterando a ordem dos parâmetros
        const sumItems = pipe(
            getItemsFromNotas,
            partialize(filterItemsByCode, '2143'),
            sumItemsValue,
        );
        return this.listAll().then(sumItems);
    }
};
```

Excelente, mas qual abordagem utilizar já que ambas chegam ao mesmo resultado? A escolha dependerá do gosto do programador ou de sua equipe. O importante é manter-se consistente em sua escolha, garantindo um padrão.

Ao longo do projeto, o autor utilizará a função `pipe`, simplesmente por ele já ter desenvoltura com o operador `|` da plataforma Linux.

Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part1/03

# Parte 2 - O último ensinamento

*O velho cangaceiro havia ensinado coisas nunca imaginadas pelo jovem cangaceiro, mas nenhuma delas foi tão transformadora quanto a ideia de criar suas próprias ferramentas combinando recursos que estão ao seu redor. Onde havia escassez, passou a haver oportunidade. Meses se passaram até que o velho cangaceiro anunciara que seu fim estava próximo. Colocando em prática a capacidade adquirida de ouvir, o jovem cangaceiro apenas observava o velhote. Ele dizia que faltava um último ensinamento. Foi então que ele disse: "A verdadeira paz não está em um grupo ou um lugar, ela está...". Sem que pudesse terminar sua fala, sua hora havia chegado. Desolado e sem saber que rumo tomar, o jovem cangaceiro pelo mundo mais uma vez se pôs a andar.*

CAPÍTULO 4

# LIMITANDO OPERAÇÕES

> *"Deus é paciência. O contrário é o diabo."* - Grande Sertão: Veredas

No capítulo anterior, aprendemos a organizar nosso código por meio da composição de funções, inclusive criamos funções utilitárias para nos auxiliar na composição. Desta vez, vamos focar nas interações do usuário.

Não é raro o desenvolvedor precisar limitar determinadas operações disparadas pelo usuário, geralmente para evitar processamento desnecessário. É por isso que criaremos dois novos requisitos da aplicação para que tenhamos uma base para praticar.

O primeiro é permitir que o usuário só execute a operação de busca de notas fiscais clicando no botão no máximo três vezes. O segundo é filtrar a quantidade de cliques do usuário dentro de uma janela de tempo.

## 4.1 CONSTRUÇÃO DA FUNÇÃO TAKEUNTIL

Como todo bom programador, vamos isolar o código que limita a execução de funções para que possamos reutilizá-lo em todas as nossas aplicações. Chamaremos a função de `takeUntil`, nome inspirado no *operator* de mesmo nome da biblioteca `RxJS` (https://github.com/Reactive-Extensions/RxJS).

Vamos declarar a função em `cangaceiro2/public/app/utils/operators.js`:

```
// cangaceiro2/public/app/utils/operators.js
// código anterior omitido

export const takeUntil = (times, fn) => {

};
```

A declaração anterior recebe dois parâmetros. O primeiro indica o número máximo de vezes que executaremos determinada operação, já o segundo é a função que representa essa operação.

A função `takeUntil` precisará retornar uma nova função configurada para chamar o parâmetro `fn` um número máximo de vezes. Lembre-se de que `fn` é a função cuja chamada desejamos limitar:

```
// cangaceiro2/public/app/utils/operators.js
// código anterior omitido

export const takeUntil = (times, fn) => {
    return () => {
        fn();
    }
};
```

Porém, do jeito como está, ela simplesmente chama a função `fn`. Que tal fazermos um teste para termos uma *big picture* de como usaremos a função `takeUntil`? Vamos lá!

Vamos alterar `cangaceiro2/public/app/app.js` e importar a função `takeUntil` para, em seguida, criar uma simples função que será chamada várias vezes:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil } from './utils/operators.js';

// código temporário de teste
const showMessage = () => console.log('Opa!');

// recebe a função que encapsula showMessage
const operation = takeUntil(3, showMessage);

// exibirá 10 mensagens, não é o que queremos!
let counter = 10;
while(counter--) operation();

// código posterior omitido
```

São exibidas 10 mensagens, com certeza não é o que queremos.

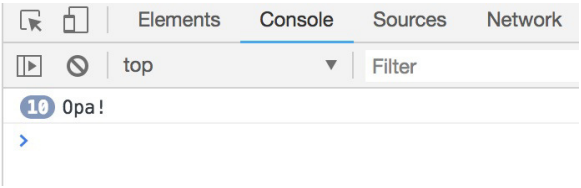



Figura 4.1: Saída no console

Vamos concluir a implementação da função `takeUntil`:

```
// cangaceiro2/public/app/utils/operators.js

export const takeUntil = (times, fn) => {
    return () => {
        if(times-- > 0) fn();
    };
```

```
};
```

A função retornada lembrará dos parâmetros recebidos pela função `takeUntil` que a retornou. Toda vez que ela for executada, verificaremos se `times--` é maior que zero.

Como usamos um pós-decremento, primeiro verificamos seu valor para depois decrementá-la. Vale lembrar que qualquer número exceto zero é avaliado como `true` em JavaScript.

Enquanto `times--` for maior que zero, chamaremos a função. Todavia, podemos simplificar nosso código ainda mais, removendo o bloco mais externo, pois só temos uma única instrução no bloco de `takeUntil`.

```
// cangaceiro2/public/app/utils/operators.js

export const takeUntil = (times, fn) =>
    () => {
        if(times-- > 0) fn();
    };
```

Podemos otimizar nosso código removendo o bloco da função retornada. No entanto, nosso código não será válido:

```
// cangaceiro2/public/app/utils/operators.js

// ERRO!
export const takeUntil = (times, fn) =>
    () => if(times-- > 0) fn();
```

Só podemos remover o bloco se a *arrow function* tiver apenas uma instrução e, no caso, ela possui duas, a condição `if` e a chamada `fn`. Mas um pequeno ajuste resolverá essa questão:

```
// cangaceiro2/public/app/utils/operators.js

export const takeUntil = (times, fn) =>
    () => times-- > 0 && fn();
```

Como estamos usando uma condição `&&`, a segunda condição (a chamada de `fn`) só será executada se `times` for `true`. Mais enxuto, não?

Agora, ao recarregarmos nossa página, veremos que nosso teste exibirá apenas três mensagens no console. Podemos remover o código de teste que adicionamos para utilizar a função `takeUntil` no problema que ela deve resolver.

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil } from './utils/operators.js';

const operation1 = takeUntil(3, () =>
    service
    .sumItems('2143')
    .then(log)
    .catch(log)
);

document
.querySelector('#myButton')
.onclick = () => operation1();
```

Perfeito! Podemos clicar quantas vezes desejarmos, que a busca das notas fiscais só será realizada no máximo três vezes:

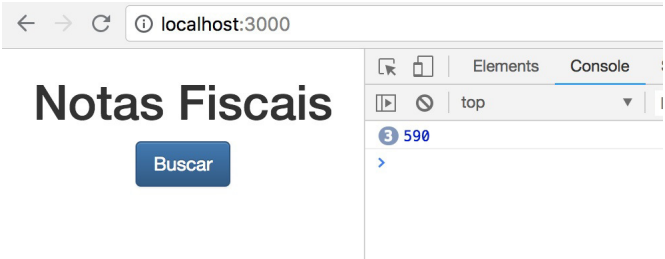



Figura 4.2: Saída no console

Todavia, podemos simplificar nosso código ligeiramente. Vejamos a linha:

```
// cangaceiro2/public/app/app.js
// código anterior omitido

// revendo o trecho de código
document
.querySelector('#myButton')
.onclick = () => operation1();
```

Nela, associamos à propriedade `onclick` uma função anônima declarada com *arrow function* que, ao ser chamada pelo clique do botão, chamará `operation1` . Ela poderia ser escrita assim:

```
// cangaceiro2/public/app/app.js
// código anterior omitido

document
.querySelector('#myButton')
.onclick = event => operation1();
```

Todavia, não temos interesse em trabalhar com o parâmetro `event` e por isso podemos atribuir diretamente `operation1` à propriedade `onclick` :

```
// cangaceiro2/public/app/app.js
// código anterior omitido

// modificando o código!
document
.querySelector('#myButton')
.onclick = operation1;
```

Não se preocupe com o nome `operation1` , ele foi utilizado para que possamos ver com clareza cada operação que criarmos, inclusive a combinação que faremos entre elas. Aliás, chegou a hora de implementarmos mais uma operação.

## 4.2 CONSTRUÇÃO DA FUNÇÃO DEBOUNCETIME

Desta vez, precisaremos filtrar a quantidade de cliques do usuário dentro de uma janela de tempo. Por exemplo, se o usuário clicar diversas vezes no botão dentro da janela de tempo de meio segundo, nós ignoraremos todos os cliques. Apenas o clique que ultrapassar meio segundo sem que um novo clique seja realizado disparará a ação do botão. Em suma, queremos aplicar o *pattern* **Debounce**. Nossa solução também deve ser reutilizável.

Dentro de `cangaceiro2/public/app/utils/operators.js`, vamos criar a função `debounceTime`:

```
// cangaceiro2/public/app/utils/operators.js
// código anterior omitido

export const debounceTime = (milliseconds, fn) => {
    return () => {

    };
};
```

Nossa função terá uma estrutura semelhante à função `takeUntil`. O primeiro parâmetro de `debounceTime` é o tempo em milissegundos no qual só pode haver uma chamada de função. Já o segundo parâmetro é a função que respeitará a janela de tempo definida.

Sabemos que podemos postergar a execução de uma função por meio da função `setTimeout`. Vamos utilizá-la:

```
// cangaceiro2/public/app/utils/operators.js
// código anterior omitido

export const debounceTime = (milliseconds, fn) => {
```

```
    return () => {
        setTimeout(fn, milliseconds);
    };
};
```

A função `debounceTime` ainda não está pronta, porém vamos testá-la em um exemplo de escopo menor.

Vamos alterar o módulo `cangaceiro2/public/app/app.js`. Nele, importaremos `debounceTime` e logo em seguida faremos um teste:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime } from './utils/operators.js';

const showMessage = () => console.log('Opa!');
const operation2 = debounceTime(500, showMessage);
operation2();
operation2();
operation2();
// código posterior omitido
```

Nesse exemplo, como estamos executando três chamadas da função `operation2` na sequência, o esperado é que apenas a última função fosse chamada, mas como ainda não terminamos a função `debounceTime`, todas são chamadas:

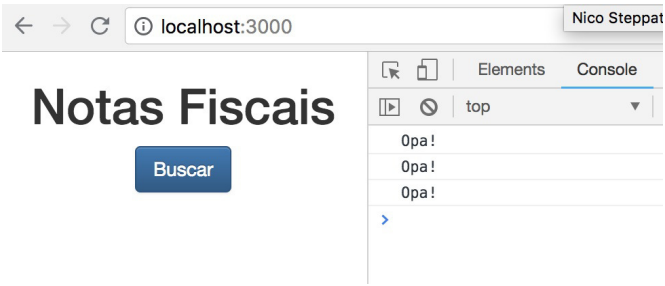



Figura 4.3: Saída no console

Nosso código posterga em meio segundo a execução de cada função. Precisamos que, assim que a função for chamada novamente, ela cancele o *timer* anterior, agendando um novo. Dessa forma, podemos executar 1000 vezes a mesma função, que apenas a última será executada quando o tempo de meio segundo expirar:

```
// cangaceiro2/public/app/utils/operators.js
// código anterior omitido

export const debounceTime = (milliseconds, fn) => {
    let timer = 0;
    return () => {
        clearTimeout(timer);
        timer = setTimeout(fn, milliseconds);
    };
};
```

Sempre que a função retornada por `debounceTime` for chamada, ela cancelará o agendamento da chamada da função `fn` por meio de `clearTimeout(timer)`. A variável `timer` começa em 0, isto é, um *timer* que não existe, mas que não dará erro algum ao ser chamado por `clearTimeout`. Depois de cancelarmos, atribuímos um novo valor a `timer` por meio do retorno de `setTimeout`. Mais uma vez a *closure* vem nos ajudar, porque a

função retornada por `debounceTime` lembrará do estado desta variável, atualizando-a sempre que necessário.

Agora nosso teste funciona como esperado, exibindo apenas o resultado da chamada da função. Vamos remover nosso teste e utilizar nossa recém-criada função:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime } from './utils/operators.js';

const operation1 = takeUntil(3, () =>
    service
    .sumItems('2143')
    .then(log)
    .catch(log)
);

const operation2 = debounceTime(500, operation1);

document
.querySelector('#myButton')
.onclick = operation2;
```

Se clicarmos freneticamente no botão e pararmos durante meio segundo, a função `takeUntil` será chamada e executará nossa operação.

Podemos ainda combinar as duas funções, evitando a declaração de mais uma variável:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime } from './utils/operators.js';
```

```
const action = debounceTime(500, takeUntil(3, () =>
    service
    .sumItems('2143')
    .then(log)
    .catch(log)
));

document
.querySelector('#myButton')
.onclick = action;
```

Excelente! Mas será que podemos utilizar a função `pipe` com as funções `takeUntil` e `debounceTime`?

## 4.3 MAIS COMPOSIÇÃO

Se quisermos realizar composição com auxílio da nossa função `pipe` precisaremos adequar essas funções para que recebam um parâmetro, apenas.

Vamos recordar a implementação das funções:

```
// cangaceiro2/public/app/utils/operators.js
// recordando o código, apenas

// código anterior omitido
export const takeUntil = (times, fn) =>
    () => times-- > 0 && fn();

export const debounceTime = (milliseconds, fn) => {
    let timer = 0;
    return () => {
        clearTimeout(timer)
        timer = setTimeout(fn, milliseconds);
    };
};
// código posterior omitido
```

As funções retornam uma função configurada que será utilizada pela nossa aplicação. Vejamos um exemplo de uso isolado

para refrescarmos nossa memória:

```
// apenas exemplo, não entra em nosso programa
const doTake = takeUntil(2, () => console.log('Chamou callback!')
);
doTake(); // exibe log
doTake(); // exibe log
doTake(); // não exibe log

const doDebounce = debounceTime(500, () => console.log('Chamou ca
llback'));
doDebounce(); // não exibe log
doDebounce(); // não exibe log
doDebounce(); // exibe log
```

Por meio da função `partialize`, podemos criar uma nova função que por baixo dos panos assumirá o primeiro parâmetro. Dessa maneira, a nova função receberá apenas o callback como parâmetro.

```
const partialTake = partialize(takeUntil, 2);

// só recebe agora um parâmetro, o callback com a lógica que será

// executada quando a função for chamada
const configuredPartialTake = partialTake(() => console.log('oi')
);

// só recebe agora um parâmetro, o callback com a lógica que será

// executada quando a função for chamada
const partialDebounce = partialize(debounceTime, 500);

// partialDebounce recebe o take já configurado
const configuredPartialDebounce = partialDebounce(configuredParti
alTake);

configuredPartialDebounce();
configuredPartialDebounce();
configuredPartialDebounce(); // exibe oi uma vez apenas
```

Agora que já recapitulamos, vamos tentar compor as

operações:

```
// cangaceiro2/public/app/app.js

import { log } from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './util
s/operators.js';

const operations = pipe(
    partialize(debounceTime, 500),
    partialize(takeUntil, 3)
);

const action = operations(() =>
    service
    .sumItems('2143')
    .then(log)
    .catch(log)
);

document
.querySelector('#myButton')
.onclick = action;
```

A tentativa foi boa, mas infelizmente um teste demonstra rapidamente que a operação não foi executada como esperado. Qual o motivo?

Um olhar atento perceberá que a operação `takeUntil` foi aplicada antes de `debounceTime`. Se isso é realmente verdade, se invertermos a ordem das funções passadas para `pipe`, o problema será resolvido? Vamos tentar:

```
// cangaceiro2/public/app/app.js
// código anterior omitido

// inverteu a ordem
const operations = pipe(
    partialize(takeUntil, 3),
```

```
    partialize(debounceTime, 500)
);

// código posterior omitido
```

Realmente, a aplicação volta a funcionar como esperado. Mas por que tivemos que inverter a ordem? Vamos escrutinar mais uma vez a função `pipe`:

```
// app/utils/operators.js
// código anterior omitido

// revendo o código apenas
export const pipe = (...fns) => value =>
    fns.reduce((previousValue, fn) =>
        fn(previousValue), value);

// código posterior omitido
```

A função `pipe` retorna uma função que, ao ser chamada, invocará cada função recebida como parâmetro da direta para esquerda, passando o resultado da função anterior para a próxima e assim sucessivamente, retornando no final o último resultado.

Na primeira iteração, teremos o retorno de `debounceTime`. Seu retorno será o callback de `takeUntil` e a função configurada de `takeUntil` será o resultado do `reduce` e, por conseguinte, o retorno de `Pipe`. Não é isso que queremos, queremos exatamente o contrário, por isso foi necessário inverter a ordem.

Esse detalhe é necessário porque por debaixo dos panos estamos utilizando a função `pipe` para encadear chamadas de callbacks.

Excelente, aprendemos a limitar determinadas ações do usuário, mas há um fator importante em toda aplicação sobre o qual não temos muito controle e que precisa de uma atenção

especial: a rede!

> Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part2/04

CAPÍTULO 5

# LENTIDÃO NA REDE

> *"Vem, vamos embora que esperar não é saber. Quem sabe faz a hora, não espera acontecer."* - Pra não dizer que não falei das flores, Geraldo Vandré

Nossa aplicação está cada vez mais recheada de recursos que melhoram a vida do usuário e do desenvolvedor. No entanto, como nossa aplicação se comportará em uma rede lenta, limitada?

## 5.1 SIMULANDO LENTIDÃO NA REDE NO NAVEGADOR

Podemos simular o comportamento de uma rede lenta por meio do Chrome Developer Tools. Faremos isso agora!

Com a aplicação carregada no navegador, vamos abrir o console do Chrome. Nele, há a aba *Network* (ou *Rede*, em português). Clicando nela, será exibida uma barra de ferramentas, na qual vamos até a opção *online*. Quando clicada, podemos escolher diferentes tipos de redes, inclusive a opção *offline*. Vamos escolher *Slow 3G*.

Quando escolhemos uma configuração de rede diferente de *online*, ao lado da aba *Network* aparecerá um aviso em amarelo, deixando claro para o desenvolvedor que a conexão do Chrome foi modificada intencionalmente.

Tendo certeza de que estamos com a conexão limitada, vamos clicar no botão e importar notas. Haverá um hiato considerável até que a operação termine. Em uma rede ainda mais lenta, esse tempo pode passar de um minuto.

Não queremos deixar o usuário esperar tanto tempo para que sua operação seja realizada. Precisamos cancelá-la dentro de um tempo que achamos justo para que o usuário possa realizá-la novamente.

A má notícia é que não existe um mecanismo de *timeout* na especificação de Promises, e se quisermos esse recurso precisaremos implementá-lo, mas como? É isso que veremos na próxima seção.

## 5.2 A FUNÇÃO PROMISE.RACE

Apesar de a especificação `Promise` não suportar um mecanismo de `timeout`, podemos implementá-lo com auxílio da função **Promise.race**. Para que possamos ver esse recurso em ação, vamos realizar um teste em `cangaceiro2/public/app/app.js`, logo abaixo das importações de módulos:

```
// cangaceiro2/public/app/app.js
// importações de módulos omitidas

const promise1 = new Promise((resolve, reject) =>
    setTimeout(() => resolve('promise 1 resolved'), 3000));
```

```
const promise2 = new Promise((resolve, reject) =>
    setTimeout(() => resolve('promise 2 resolved'), 1000));

// código posterior omitido
```

Temos duas `Promises` declaradas. A primeira só será resolvida depois de três segundos, já a segunda, depois de um segundo. Todavia, queremos realizar uma "corrida" (*race*) entre as Promises. Nessa corrida, só queremos o resultado daquela que for resolvida primeiro. É aí que entra `Promise.race`.

A função `Promise.race` recebe uma lista de Promises e, assim que uma delas for resolvida, receberemos imediatamente seu resultado na próxima chamada encadeada à `then`. As demais Promises são ignoradas:

```
// cangaceiro2/public/app/app.js

// importações de módulos omitidas

const promise1 = new Promise((resolve, reject) =>
    setTimeout(() => resolve('promise 1 resolvida'), 3000));

const promise2 = new Promise((resolve, reject) =>
    setTimeout(() => resolve('promise 2 resolvida'), 1000));

// exibirá no console "promise 2 resolvida";
Promise.race([
    promise1,
    promise2
])
.then(log)
.catch(log);

// código posterior omitido
```

É importante estar atento a qualquer rejeição que aconteça durante a resolução das Promises. Rejeições direcionarão o fluxo da aplicação para dentro da função `catch` da Promise:

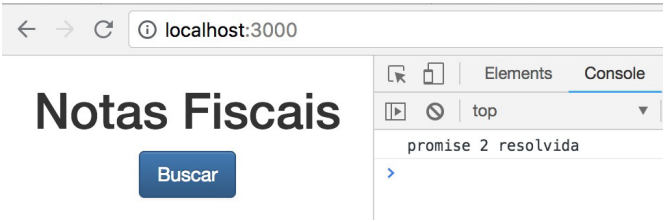



Figura 5.1: Saída no console

Em suma, estamos interessados no resultado da primeira Promise resolvida, mas se algum erro acontecer antes de qualquer resultado válido, cairemos dentro da função `catch`.

Agora que já entendemos a mecânica de `Promise.race`, podemos remover o teste que fizemos e partir para a implementação do nosso mecanismo de `timeout`.

## 5.3 IMPLEMENTANDO TIMEOUT EM PROMISES

Vamos criar a função `timeoutPromise` no módulo `cangaceiro2/public/app/utils/promise-helpers.js`:

```
// cangaceiro2/public/app/utils/promise-helpers.js

// código anterior omitido

export const timeoutPromise = (milliseconds, promise) => {

};
```

Nossa função recebe a Promise que desejamos executar e o tempo do `timeout`. Internamente, ela declarará uma nova Promise que será resolvida quando o tempo do `timeout` expirar:

```
// cangaceiro2/public/app/utils/promise-helpers.js
```

```
export const timeoutPromise = (milliseconds, promise) => {
    const timeout =  new Promise((resolve, reject) =>
        setTimeout(() =>
            reject(`Limite da promise excedido (limite: ${millise
conds} ms)`),
                milliseconds));
};
```

Agora, basta utilizarmos `Promise.race` passando como parâmetro a Promise recebida por `timeoutPromise` e aquela que representa o `timeout`:

```
// cangaceiro2/public/app/utils/promise-helpers.js

export const timeoutPromise = (milliseconds, promise) => {
    const timeout =  new Promise((resolve, reject) =>
        setTimeout(() =>
            reject(`Limite da operação excedido (limite: ${millis
econds} ms)`),
                milliseconds));

    return Promise.race([
        timeout,
        promise
    ]);
};
```

Excelente! Se a nossa Promise for resolvida primeiro, não haverá `timeout`, porém, se `timeout` ganhar a corrida, ele forçará um `reject`.

Vamos alterar o módulo `cangaceiro2/public/app/app.js` para fazer uso da nossa `timeoutPromise`:

```
// cangaceiro2/public/app/app.js
// importou timeoutPromise

import { log, timeoutPromise} from './utils/promise-helpers.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './util
```

```
s/operators.js';

const operations = pipe(
    partialize(takeUntil, 3),
    partialize(debounceTime, 500),
);

// usando timeoutPromise
const action = operations(() =>
    timeoutPromise(200, service.sumItems('2143'))
    .then(log)
    .catch(log)
);

document
.querySelector('#myButton')
.onclick = action;
```

O tempo de um segundo foi intencional para facilitar nossos testes. Para que vejamos a mensagem de *timeout* sendo exibida no console, precisamos alterar a rede para `Slow 3G` .

## 5.4 REPETINDO OPERAÇÕES

Não são raros locais nos quais a internet é intermitente, inclusive em áreas com rede de alta velocidade. Acessar a internet dentro de um elevador, dentro de uma embarcação ou até mesmo no estacionamento de um shopping pode contribuir para a instabilidade da rede. Existem diferentes estratégias para se lidar com a intermitência aqui descrita.

Uma aplicação pode funcionar temporariamente offline para mais tarde sincronizar as operações do usuário. A aplicação pode até mesmo realizar novamente a operação um certo número de vezes dentro de um espaço de tempo para só considerar a operação fracassada depois que todas as tentativas tiverem sido esgotadas. É

essa solução que será abordada a seguir.

## 5.5 IMPLEMENTANDO DELAY EM PROMISES

Promises não possuem nativamente um mecanismo de *retry*, motivo pelo qual teremos que implementar este recurso. Entre cada nova tentativa será preciso realizar um pequeno *delay* para darmos chance para a rede se recuperar. É por esse motivo que iniciaremos nossa implementação criando um mecanismo de *delay* que será utilizado entre chamadas de Promises.

Antes de implementarmos nossa função que permitirá realizar um *delay* entre chamadas de Promises, vamos ver como ela será utilizada em nosso código. Realizaremos um *delay* de cinco segundos:

```
// cangaceiro2/public/app/app.js
// código anterior omitido
// exemplo de uso apenas, falta implementar delay
const action = operations(() =>
    timeoutPromise(200, service.sumItems('2143'))
    .then(delay(5000)) // chamou delay
    .then(log)
    .catch(log)
);

// código posterior omitido
```

Nessa prova de conceito, basta encadearmos uma chamada da função `delay` para que a próxima chamada à `then()` seja postergada. A função recebe como parâmetro o tempo da espera em milissegundos. Todavia, ela deve ser capaz de receber o resultado da chamada anterior e passá-la para o próximo `then()` encadeado. Sem isso, não seremos capazes de obter o total dos itens na chamada `then(log)` .

Vamos implementar a função `delay` no módulo `cangaceiro2/public/app/utils/promise-helpers.js`:

```
// cangaceiro2/public/app/utils/promise-helpers.js
// código anterior omitido

export const delay = milliseconds => data =>
    new Promise(resolve =>
        setTimeout(() => resolve(data), milliseconds)
    );
```

A função `delay` recebe como parâmetro o tempo em milissegundos do *delay* e possui como retorno uma nova função que lembrará do tempo recebido pela função mais externa (*closure* em ação!). A nova função recebe apenas um parâmetro, que chamamos de `data`, **dado** em português. Esse parâmetro é importante, pois é por meio dele que podemos passar o valor anterior à chamada de `delay` para a próxima chamada encadeada à `then`. Por fim, ao ser invocada, a função retornará uma nova Promise que será resolvida com uma chamada de `setTimeout`. Quando resolvida, passará o valor recebido da chamada `then()` anterior para sua próxima chamada encadeada.

Vamos alterar `app/app.js` para importar e utilizar nossa função. Nada mais justo do que realizarmos um teste:

```
// cangaceiro2/public/app/app.js

// importou delay!
import { log, timeoutPromise, delay} from './utils/promise-helper
s.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './util
s/operators.js';

const operations = pipe(
    partialize(takeUntil, 3),
```

```
    partialize(debounceTime, 500),
);

const action = operations(() =>
    timeoutPromise(200, service.sumItems('2143'))
    .then(delay(5000)) // chamou delay
    .then(log)
    .catch(log)
);

document
.querySelector('#myButton')
.onclick = action;
```

Excelente! O total será exibido no console cinco segundos após termos clicado no botão.

Agora que já temos nossa função `delay` pronta, chegou a hora de implementarmos nosso mecanismo de *retry*.

## 5.6 IMPLEMENTANDO RETRY EM PROMISES

A função `retry` receberá uma função que, ao ser chamada, deve retornar uma nova Promise com a operação que desejamos realizar, o número de tentativas e o intervalo de tempo entre essas tentativas.

Vamos criar seu esqueleto em `cangaceiro2/public/app/utils/promise-helpers.js`:

```
// cangaceiro2/public/app/utils/promise-helpers.js

// código anterior omitido

export const retry = (retries, milliseconds, fn) => // falta impl
ementar
```

A primeira coisa que faremos é chamar a função `fn` e

programar uma resposta no caso de sua rejeição, isto é, caso algum erro aconteça durante sua execução. Sabemos que é por meio da função `catch` que lidamos com exceções.

Como precisaremos repetir a operação até atingirmos o limite definido pelo parâmetro `retries`, faz bastante sentido lançarmos mão de recursão.

Em outras palavras, nossa função `retry` será uma função recursiva, que nada mais é do que uma função que chama a si mesma até que um *leave event* ocorra, caso contrário cairemos em uma recursão infinita que travará nossa aplicação:

```
// cangaceiro2/public/app/utils/promise-helpers.js
// código anterior omitido

export const retry = (retries, milliseconds, fn) =>
    fn().catch(err => {
        console.log(retries);
        return retries > 1
            ? retry(--retries, milliseconds, fn)
            : Promise.reject(err));
    });
```

Na cláusula `catch`, com um `if` ternário, testamos se o número de tentativas ainda é maior do que um (está certo, porque já gastamos uma tentativa na chamada da Promise). Se for, temos direito a mais uma tentativa e chamamos recursivamente `retry(fn, --retries, time)`. Se o número máximo de tentativas for excedido, retornamos uma rejeição com `Promise.reject(err)`, que recebe a causa do último erro.

O que as chamadas recursivas farão é encadear uma sucessão de chamadas à `then()`, repetindo a operação. Faz sentido, pois precisamos de uma nova Promise a cada tentativa.

Todavia, é preciso haver um intervalo entre as tentativas. Já temos a função `delay` e só nos resta combiná-la com `retry`:

```
// cangaceiro2/public/app/utils/promise-helpers.js
// código anterior omitido

export const retry = (retries, milliseconds, fn) =>
    fn().catch(err => {
        console.log(retries);
        return delay(milliseconds)().then(() =>
            retries > 1
                ? retry(--retries, milliseconds, fn)
                : Promise.reject(err))
    });
```

Foi necessário realizar `delay(time)()`, porque a função `delay` retorna uma nova função que, ao ser chamada, devolve uma Promise e, como já vimos, a chamada encadeada à `then()` só será feita depois de o tempo do *delay* ter expirado.

Agora que já temos nossa função pronta, vamos utilizá-la em `cangaceiro2/public/app/app.js`:

```
// cangaceiro2/public/app/app.js

// importação de delay removida e em seu lugar importamos retry
import { log, timeoutPromise, retry} from './utils/promise-helper
s.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './util
s/operators.js';

const operations = pipe(
    partialize(takeUntil, 3),
    partialize(debounceTime, 500),
);

// utilizando retry
const action = operations(() =>
    retry(3, 3000, () => timeoutPromise(200, service.sumItems('21
```

```
43')))
    .then(log)
    .catch(log)
);

document
.querySelector('#myButton')
.onclick = action;
```

Para testarmos, basta carregar a página no navegador, parar o servidor e clicar no botão. Veremos através do console as mensagens das tentativas que serão realizadas dentro de um intervalo de cinco segundos. Na segunda tentativa, podemos subir rapidamente o servidor para que a operação seja realizada com sucesso:

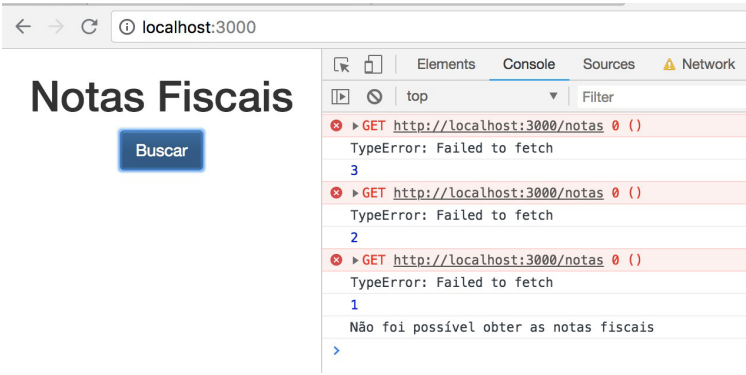



Figura 5.2: Saída no console

Desde o início do projeto, focamos na manutenção e legibilidade do nosso código. Tratando-se de manutenção, ainda há mais uma melhoria que pode ser feita, e ela será o assunto do próximo capítulo.

Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part2/05

Capítulo 6

# COMBATENDO O ALTO ACOPLAMENTO

> *"Solta os presos, que o mundo já é uma prisão."* - Lampião

Nossa aplicação está exibindo a soma dos itens das notas no console, nada extraordinário. Mas e se além de fazermos isso quisermos exibir um `alert` com o total calculado? Teremos que alterar a `app.js` e adicionar toda a lógica interessada na soma dos itens na instrução `then`:

```
// cangaceiro2/public/app/app.js
// código anterior omitido

const action = operations(() =>
    retry(3, 3000, () => timeoutPromise(200, service.sumItems('21
43')))
    .then(soma => {
        console.log(soma); // um interessado
        alert(soma); // outro interessado
    })
    .catch(log)
);

// código posterior omitido
```

Podemos ter mais de um interessado no total das notas, por exemplo, para renderizar a resposta do usuário e assim por diante. Para cada novo interessado, teremos que alterar o módulo `app.js`, correndo o risco de quebrá-lo.

Em suma, o módulo `app.js` terá mais de um motivo para ser alterado. Uma solução para o alto acoplamento aqui citado é o emprego do *pattern* **Publish-Subscribe**.

## 6.1 O PATTERN PUBLISH-SUBSCRIBE

*Publish/Subscribe* é um padrão de projeto que procura reduzir o acoplamento de uma aplicação evitando que o emissor (*publisher*) e receptores (*subscribers*) troquem mensagens diretamente em si. Por exemplo, se um emissor envia a mensagem diretamente para 10 receptores, ele estará acoplado a todos eles.

Como solução, toda comunicação entre as partes interessadas é feita por meio de um barramento muitas vezes chamado de *Broker*. Este barramento recebe mensagens dos *publishers* e as reencaminha para *subscribers* cadastrados. Com este padrão, *publishers* e *subscribers* ficarão acoplados apenas ao barramento. Aliás, o barramento costuma ter dois métodos:

- *publish(topico, dado)*: notifica todos que estejam inscritos ao tópico passado como primeiro parâmetro. O segundo parâmetro é qualquer informação que queiramos fornecer para os inscritos.
- *subscribe(topico)*: realiza a inscrição para um tópico. É possível haver um ou mais inscritos e todos eles serão notificados toda vez que houver uma publicação para este

tópico.

No entanto, utilizamos outra nomenclatura, a mesma utilizada pela plataforma Node.js.

## 6.2 SOBRE EVENTEMITTER

Nessa plataforma, há a classe `EventEmitter` que implementa o *pattern Publish-Subscribe*. Ele possui os métodos `emit` e `on` que equivalem respectivamente aos métodos `publish` e `subscribe`. Em suma, faremos um simples clone do `EventEmitter` para o browser.

No contexto do nosso problema, queremos no final um código nesta estrutura:

```
// app/app.js
// código anterior omitido

// apenas exemplo, ainda não entra em nosso código
const action = operations(() =>
    retry(3, 3000, () => timeoutPromise(1000, service.sumItems('2
143')))
    .then(total => EventEmitter.emit('itensTotalizados', total))
    .catch(log)
);

// código posterior omitido
```

Reparem que não há mais a chamada de `.then(sumItemsWithCode)`, pois todo o código que lida com as notas também será isolado de `app/app.js`.

Se tivermos outros interessados no tópico `itensTotalizados`, basta que eles se inscrevam neste tópico para serem notificados toda vez que houver uma publicação. Vejamos

um exemplo:

```
// exemplo apenas

// módulo A
EventEmitter.on('itensTotalizados', total => console.log(total);

// módulo B
EventEmitter.on('itensTotalizados', total => alert(total);
```

Os módulos `A` e `B` executaram operações distintas para uma mesma publicação.

Agora que já temos uma visão geral do *pattern*, vamos implementá-lo em nosso projeto.

## 6.3 IMPLEMENTANDO UM EVENTEMITTER

Vamos criar o módulo `app/utils/event-emitter.js`. Utilizaremos um `Map` para associar um *event* a todos os seus *listeners*. Como a chave do `Map` será o `event`, não corremos o risco de criar uma nova chave em nosso `Map` para um `event` já existente:

```
// app/utils/event-emitter.js

const events  = new Map();
```

Pronto. Um ponto a destacar é que `events` não será uma propriedade do objeto `EventEmitter` que criaremos. A variável `events` vive no escopo do módulo. Essa abordagem evitará que outra parte do nosso código, que não seja o próprio `EventEmitter`, interaja e bagunce com `events`. Em outras palavras, estamos encapsulando o mapa de eventos.

Vamos exportar o objeto `EventEmitter` que terá os métodos

`on` e `emit`:

```
// app/utils/event-emitter.js

const events  = new Map();

export const EventEmitter = {

    on(event, listener) {
        /* falta implementar */
    },

    emit(event, data) {
        /* falta implementar */
    }
};
```

Vamos começar pela implementação do método `on`. A primeira coisa que ele fará é verificar se o `event` que desejamos escutar já existe. Se não existir, criamos uma chave em nosso `Map` com o nome do `event` e guardamos um array vazio que armazenará todos os seus `listeners`. É a partir dessa garantia que podemos adicionar o `listener` ao `event`:

```
// app/utils/event-emitter.js

const events  = new Map();

export const EventEmitter = {

    on(event, listener) {
        if (!events.has(event)) events.set(event, []);
        events.get(event).push(listener);
    },

    emit(event, data) {
        /* falta implementar */
    }
};
```

Só falta o método `emit`. Quando for chamado, ele obterá a

lista de `listeners` para o `event` passado como parâmetro. Se o `event` ainda não foi criado por nenhum `listener`, nada acontecerá.

Cada `listener` associado ao `event` será chamado recebendo como parâmetro o dado recebido pela função `emit`. Esse passo é fundamental, caso contrário os `listeners` não terão acesso ao dado associado com o `event`:

```
// app/utils/event-emitter.js

const events  = new Map();

export const EventEmitter = {

    on(event, listener) {
        if (!events.has(event)) events.set(event, []);
        events.get(event).push(listener);
    },

    emit(event, data) {
        const listeners = events.get(event);
        if (listeners) listeners
            .forEach(listener => listener(data));
    }
};
```

Chegou a hora de utilizarmos nosso `EventEmitter`.

## 6.4 DESACOPLANDO NOSSO CÓDIGO

A primeira alteração que faremos é importar no módulo `app/app.js` nosso `EventEmitter` para que possamos utilizá-lo. Em seguida, no lugar de imprimirmos o total calculado no console, dispararemos o evento `itensTotalizados` por meio da função `EventEmitter.emit()`, com o total calculado:

```
// cangaceiro2/public/app/app.js
```

```
import { log, timeoutPromise, retry} from './utils/promise-helper
s.js';
import './utils/array-helpers.js';
import { notasService as service } from './nota/service.js';
import { takeUntil, debounceTime, partialize, pipe } from './util
s/operators.js';

// importou
import { EventEmitter } from './utils/event-emitter.js';

const operations = pipe(
    partialize(takeUntil, 3),
    partialize(debounceTime, 500),
);

// disparando o evento com o seu respectivo dado
const action = operations(() =>
    retry(3, 3000, () => timeoutPromise(1000, service.sumItems('2
143')))
    .then(total => EventEmitter.emit('itensTotalizados', total))
    .catch(log)
);

document
.querySelector('#myButton')
.onclick = action;
```

Vamos criar os módulos `app/alert-handler.js` e `app/console-handler.js`, respectivamente, aqueles que escutarão ao evento `itensTotalizados` e responderão adequadamente a ele.

Vamos começar por `alert-handler.js`, que apenas exibirá o total recebido através de um alerta.

```
// app/alert-handler.js

import { EventEmitter } from './utils/event-emitter.js';

EventEmitter.on('itensTotalizados', alert);
```

Agora vamos para `console-handler.js`, aquele que exibirá o total recebido no console.

```
// app/console-handler.js

import { EventEmitter } from './utils/event-emitter.js';

EventEmitter.on('itensTotalizados', console.log);
```

Vamos importá-los independente do módulo `app.js`:

```
    <!-- cangacero2/public/index.html -->
    <!-- código anterior omitido -->
    <script type="module" src="app/app.js"></script>
    <script type="module" src="app/alert-handler.js"></script>

    <script type="module" src="app/console-handler.js"></script>

</body>
</html>
```

Isso já é o suficiente para que nosso alerta dispare e a informação do total seja exibida no log do navegador.

Em nenhum momento `console-handler.js` ou `alert-handler.js` souberam da existência de `app.js` e nem este módulo dos outros dois scripts. Toda comunicação foi feita por meio de `EventEmitter`, que foi importado em todos eles.

> Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part2/06

Capítulo 7

# A MÔNADA MAYBE

> *"Nada jamais continua, tudo vai recomeçar!"* - Mário Quintana

Consumimos os dados de uma API, excelente. Nem sempre temos controle sobre a API que consumimos e por isso precisamos nos blindar de possíveis problemas que possam ocorrer. Tratando-se do imprevisível, o que aconteceria se nossa API retornasse `null` como resposta? Sabemos que este tipo de dado pode causar problemas em métodos e funções que não estejam preparados para lidar com este valor.

Vamos simular um retorno `null` alterando `service.js` temporariamente:

```
// app/nota/service.js
// código anterior omitido

listAll() {
    return fetch(API)
        .then(fetchHandler)
        .then(notas => null)
        // código omitido
}
```

Com a alteração feita, em `index.html` vamos clicar no único botão da página, aquele que busca as notas fiscais. Nada será exibido para o usuário, mas se escrutinarmos o console do navegador, veremos a seguinte mensagem de erro:

> *Cannot read property '$flatMap' of null*

Se traduzirmos para o português, temos "Não consigo ler a propriedade '$flatMap' de null". Vamos nos aprofundar nessa questão. Isso acontece porque o valor `null` foi passado para a composição `sumItems` , que não está preparada para lidar com este tipo de dado.

Podemos resolver o problema adicionando uma condição `if` no método `list()` de `notasService` . Ela verificará se o retorno da API é `null` e em seu lugar retornará um array vazio. O array vazio será suficiente para que nossa aplicação não quebre:

```
// apenas exemplo, não entra no código ainda

// app/nota/service.js
// código anterior omitido

listAll() {
    return fetch(API)
        .then(fetchHandler)
        .then(notas => {
            if(notas) return notas;
            return [];
        })
        // código omitido
}
```

Todavia, essa solução deixa a desejar porque precisaremos

adicionar uma condição `if` em todos os locais que talvez recebam um valor nulo, o que pesará na manutenção do nosso código.

Uma solução é criar um novo tipo, um tipo especial que encapsule o valor `null` e que seja o responsável por nos fornecer o valor encapsulado corretamente. É um tipo misterioso que criaremos gradativamente.

## 7.1 LIDANDO COM DADOS NULOS

Vamos solucionar gradativamente o problema. Vamos criar o módulo `cangaceiro2/public/app/utils/maybe.js`.

A palavra "maybe" significa "talvez" em português. Pelo menos o nome do nosso módulo faz sentido, porque talvez tenhamos um valor `null`, talvez não.

Vamos declarar no módulo que acabamos de criar a classe `Maybe` que encapsulará um valor que potencialmente pode ser `null` ou `undefined`. A ideia é que ela nos blinde de acessar esses valores:

```
// cangaceiro2/public/app/utils/maybe.js

export class Maybe {
    constructor(value) {
        this._value = value;
    }
}
```

Em `app/app.js`, vamos realizar alguns testes. Primeiramente, vamos criar duas instâncias de `Maybe`. A primeira encapsulará o valor `10` e a segunda, o valor `null`:

```
// cangaceiro2/public/app/app.js

// importações anteriores omitidas
import { Maybe } from './utils/maybe.js';

const maybe1 = new Maybe(10);
const maybe2 = new Maybe(null);
```

Criamos duas instâncias de `Maybe`. Todavia, podemos facilitar ligeiramente a vida do programador ao criar este tipo, evitando-o de utilizar operador `new`. Vamos adicionar o método estático `of` à classe. É este método que receberá o valor que será encapsulado e também será o responsável por invocar o operador `new`, isto é, será o responsável por criar a instância de `Maybe`:

```
// cangaceiro2/public/app/app.js

export class Maybe {

    constructor(value) {
        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }
}
```

Vamos utilizar o novo método em `cangaceiro2/public/app/app.js`. Deixar apenas o primeiro `Maybe`, pois um apenas é suficiente para avançarmos com nosso entendimento:

```
// cangaceiro2/public/app/app.js

// importações omitidas
import { Maybe } from './utils/maybe.js';

const maybe = Maybe.of(10);
```

Nosso `Maybe` tem como objetivo nos blindar de acessar um valor `null` ou `undefined`. Para isso, ele precisa ter algum método que nos indique se há valor ou não. Vamos criar o método `isNothing`:

```
// cangaceiro2/public/app/utils/maybe.js

export class Maybe {

    constructor(value) {
        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }
}
```

Usando o novo método:

```
// cangaceiro2/public/app/app.js
// importações omitidas

import { Maybe } from './utils/maybe.js';

const maybe = Maybe.of(10);
if(!maybe.isNothing()) {
    // acessa o valor, mas como, se ele está encapsulado?
}
```

Essa abordagem ainda não está muito legal porque estamos, em primeiro lugar, fazendo um `if` então cairemos no problema anterior de termos que ficar adicionando várias condições `if` ao acessar o valor em todos os lugares em que ele for chamado.

Em segundo lugar, se quisermos acessar o dado do `Maybe`,

precisaremos acessar a propriedade `_value`. Todavia, sabemos que propriedades e métodos prefixados com *underline* não devem ser acessados fora da definição da classe:

```
// cangaceiro2/public/app/app.js
// importações omitidas

import { Maybe } from './utils/maybe.js';

const maybe = Maybe.of(10);
if(!maybe.isNothing()) {
    // não deveria fazer isso!
    alert(maybe._value);
}
```

E agora? Se nosso `Maybe` é um *wrapper*, que embrulha/encapsula um valor, podemos realizar uma operação `map`. E o que é uma operação `map`, mesmo? É pegar um dado encapsulado ou armazenado e aplicar uma lógica de transformação nele.

Lembram do `Array.map`? É a mesma lógica, o `Array` encapsula uma lista de dados e aplica um `map` sobre eles realizando uma transformação.

Em nosso caso, nosso `Maybe` encapsula um valor apenas, e por meio de um `map` podemos realizar transformações nele:

```
// cangaceiro2/public/app/utils/maybe.js

export class Maybe {

    constructor(value) {
        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }
```

```
    isNothing() {
        return this._value === null || this._value === undefined;
    }

    map(fn) {
        if(this.isNothing()) return Maybe.of(null);
        const value = fn(this._value);
        return Maybe.of(value);
    }
}
```

O método `map` recebe como parâmetro uma função que terá acesso ao valor encapsulado pela `Maybe`. A primeira coisa que o método fará é verificar se o valor capsulado é `null` ou `undefined`, por meio do método `isNothing`. Se for um dos dois valores, será retornado um novo `Maybe`, que encapsulará o valor `null`. Caso o valor seja válido, aplicaremos a função `fn` sobre o valor para, em seguida, embrulharmos o valor resultante em uma nova `Maybe` que será retornada.

Podemos simplificar o método `map` da seguinte maneira:

```
// cangaceiro2/public/app/utils/maybe.js

// código anterior omitido

map(fn) {
    if(this.isNothing()) return Maybe.of(null);
    return Maybe.of(fn(this._value));
}

// código posterior omitido
```

Reparem que passamos diretamente para a função `Maybe.of` o resultado da aplicação `fn` sobre o valor encapsulado pela mônada.

Muito bem, chegou a hora de adequarmos nosso teste. Da

maneira como estruturamos nosso `Maybe`, só podemos interagir e modificar o valor encapsulado por ele por meio da função `map`:

```
// cangaceiro2/public/app/app.js
// importações omitidas

import { Maybe } from './utils/maybe.js';

const maybe = Maybe
    .of(10)
    .map(value => value + 10)
    .map(value => value + 30);

// código posterior omitido
```

A função `map` nos dá acesso ao valor encapsulado pelo `Maybe`. Pegamos esse valor e somamos com `10`. O retorno será um novo `Maybe` que encapsula agora o valor `20`. Como o retorno é um novo `Maybe`, podemos encadear uma chamada à função `map` mais uma vez. No final, teremos como resultado um `Maybe` que encapsula o valor transformado do `Maybe` original, que será `50`.

E se passarmos `null`? Não teremos erros nas operações `map`, pois elas serão ignoradas e, no final, teremos um `Maybe` com o valor `null` encapsulado:

```
// cangaceiro2/public/app/app.js
// importações omitidas

import { Maybe } from './utils/maybe.js';

const maybe = Maybe
    .of(null)
     /*
        Não realizará os maps abaixo, simplesmente retornará
        um novo Maybe que encapsula null
    */
    .map(value => value + 10)
```

```
    .map(value => value + 30);
```

Mas como teremos acesso ao valor encapsulado pelo `Maybe`? Ele continua encapsulado. Vamos criar o método `get`, que retornará o valor encapsulado:

```
// cangaceiro2/public/app/utils/maybe.js

export class Maybe {

    constructor(value) {
        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }

    map(fn) {
        if(this.isNothing()) return Maybe.of(null);
        return Maybe.of(fn(this._value));
    }

    // novo método
    get() {
        return this._value;
    }
}
```

Agora vamos adequar nosso teste para que utilize o método `get` que acabamos de criar:

```
// cangaceiro2/public/app/app.js
// importações omitidas

import { Maybe } from './utils/maybe.js';

const resultado = Maybe
    .of(10)
```

```
    .map(value => value + 10)
    .map(value => value + 30)
    .get(); // retorna 50!

console.log(resultado); // 50
```

Excelente, mas pode ficar ainda melhor. E se o valor no final for `null`? Será que podemos fornecer um valor padrão para ser utilizado no lugar de `null`? Claro que podemos, e faremos isso alterando o nome do método `get` para `getOrElse`, para, em seguida, adequarmos sua lógica:

```
// cangaceiro2/public/app/utils/maybe.js

export class Maybe {

    constructor(value) {
        this._value = value;
    }

    static of(value) {
        return new Maybe(value);
    }

    isNothing() {
        return this._value === null || this._value === undefined;
    }

    map(fn) {
        if(this.isNothing()) return Maybe.of(null);
        const value = fn(this._value);
        return Maybe.of(value);
    }

    // método modificado
    getOrElse(value) {
        if(this.isNothing()) return value;
        return this._value;
    }
}
```

Testando o novo método em `app.js`:

```
// cangaceiro2/public/app/app.js
// importações omitidas

import { Maybe } from './utils/maybe.js';

const resultado = Maybe
    .of(null)
    .map(value => value + 10)
    .map(value => value + 30)
    .getOrElse(0); // retorna 0

console.log(resultado); // 0
```

Excelente. Agora que terminamos nossa implementação, podemos evidenciar um aspecto interessante do nosso código.

## 7.2 CRIAMOS UMA MÔNADA SEM SABER!

A classe `Maybe` que criamos no capítulo anterior é uma mônada, que nada mais é do que uma estrutura que embrulha um valor. Para interagirmos com o valor encapsulado pela mônada, utilizamos a função `map`, algo que se coaduna com a ideia de *Functor*, que já vimos. Sendo assim, podemos dizer que nossa *maybe monad* é um Functor, por natureza.

Nossa mônada brilha mais dentro do paradigma da Programação Funcional, pois ela evita a proliferação de `if` nas funções, principalmente naquelas envolvidas em composições.

Podemos ver os benefícios da nossa mônada no que diz respeito à composição de funções com um novo exemplo. Nele, faremos a composição de duas funções. A primeira converte um texto para um array, e a segunda, um array para texto:

```
// cangaceiro2/public/app/app.js
// abaixo do último teste que fizemos
```

```
const textToArray = text => Array.from(text);
const arrayToText = array => array.join('');

// código posterior omitido
```

Usando as funções encadeadas, temos:

```
// cangaceiro2/public/app/app.js
// abaixo do último teste que fizemos

const textToArray = text => Array.from(text);
const arrayToText = array => array.join('');
const result = arrayToText(textToArray('Cangaceiro'));
alert(result); // Cangaceiro

// código posterior omitido
```

Podemos inclusive utilizar a função `pipe` para realizar a composição das funções:

```
// cangaceiro2/public/app/app.js
// abaixo do último teste que fizemos

const textToArray = text => Array.from(text);
const arrayToText = array => array.join('');
const transform = pipe(textToArray, arrayToText);
const result = transform('Cangaceiro');
alert(result);

// código posterior omitido
```

Tudo funciona perfeitamente, mas se passarmos o valor `null` nossa composição quebrará. E se a função receber o tipo `Maybe` ?

```
// cangaceiro2/public/app/app.js
// abaixo do último teste que fizemos

const textToArray = text => Array.from(text);
const arrayToText = array => array.join('');
const transform = pipe(textToArray, arrayToText);
const result = transform(Maybe.of('Cangaceiro'));
alert(result);
```

```
// código posterior omitido
```

Continuaremos com problema, pois as funções `textToArray` e `arrayToText` não esperam receber o tipo `Maybe`. Vamos alterá-las:

```
// cangaceiro2/public/app/app.js
// abaixo do último teste que fizemos

const textToArray = textM => textM.map(text => Array.from(text));
const arrayToText = arrayM => arrayM.map(array => array.join(''))
;
const transform = pipe(textToArray, arrayToText);
const result = transform(Maybe.of('Cangaceiro'));
alert(result.getOrElse(''));

// código posterior omitido
```

O resultado será "Cangaceiro". Se olharmos nossas funções, em nenhum momento precisamos adicionar condições `if`.

Se passarmos um `Maybe.of(null)`, o resultado final será uma string vazia no lugar de `null`.

Agora precisamos aplicar os conhecimentos aqui adquiridos para resolver o problema da nossa aplicação.

## 7.3 UTILIZANDO NOSSO TIPO MONÁDICO

Vamos remover os testes que fizemos em `app/app.js` para, em seguida, alterarmos nossa aplicação para fazer uso da *maybe monad*.

Vamos importar `Maybe` em `app/nota/service.js`, e alterar o método `list()` para retornar uma mônada encapsulando o tipo `null`. Por fim, vamos alterar todas funções usadas na

composição de `sumItems` para que recebam o tipo monádico:

```
// cangaceiro2/public/app/nota/service.js

import { handleStatus } from '../utils/promise-helpers.js';
import { partialize, pipe } from '../utils/operators.js';
// importou Maybe
import { Maybe } from '../utils/maybe.js';

const API = 'http://localhost:3000/notas';

// adequou cada função para receber o tipo monádico

const getItemsFromNotas = notasM =>
    notasM.map(notas => notas.$flatMap(nota => nota.itens));

const filterItemsByCode = (code, itemsM) =>
    itemsM.map(items => items.filter(item => item.codigo == code)
);

const sumItemsValue = itemsM =>
    itemsM.map(items => items.reduce((total, item) => total + ite
m.valor, 0));

export const notasService = {

    listAll() {
        return fetch(API)
            .then(handleStatus)
            // retorna um `Maybe`, forçando null
            .then(notas => Maybe.of(null))
            .catch(err => {
                console.log(err);
                return Promise.reject('Não foi possível obter as
notas fiscais');
            });
    },

    sumItems(code) {
        const filterItems = partialize(filterItemsByCode, code);

        const sumItems = pipe(
            getItemsFromNotas,
            filterItems,
```

```
            sumItemsValue
        );

        return this.listAll()
            .then(sumItems)
            // obtendo o valor da mônada
            .then(result => result.getOrElse(0));
    }
};
```

Se nossa API um dia retornar um valor `null`, não teremos problemas em nossa aplicação. Não se esqueça de remover o `Maybe.of(null)` do código, pois foi apenas um teste:

```
// cangaceiro2/public/app/nota/service.js

// código anterior omitido
listAll() {
    return fetch(API)
        .then(handleStatus)
        .then(notas => Maybe.of(notas)) // alterou aqui
        .catch(err => {
            console.log(err);
            return Promise.reject('Não foi possível obter as nota
s fiscais');
        });
},
// código posterior omitido
```

Terminamos o nosso projeto, mas não pense o leitor que a nossa jornada no sertão acabou.

## 7.4 MAIS PADRÕES DE PROJETO

Um dos maiores desafios é criar um único projeto que comporte a aplicação de diversos padrões de projeto sem forçar uma barra no que diz respeito à motivação de sua aplicação. É por isso que a partir de agora seguiremos um rumo diferente.

Nos próximos capítulos, cada padrão de projeto apresentado trará sua própria problemática e solução, tornando sua aplicação e motivação mais adequadas. O leitor poderá aplicá-los no seu dia a dia, seja por meio do paradigma orientado a objetos ou pelo paradigma funcional.

> Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part2/07

# Parte 3 - Encontrando a paz

*Depois de anos em sua busca vã pelo último ensinamento do mestre, o jovem cangaceiro decidiu recapitular sua vida, revendo todos os seus erros e acertos durante sua jornada. Durante o processo, repetiu a frase do seu antigo mestre e, para seu espanto, era capaz de completá-la: "A verdadeira paz não está em um grupo ou um lugar, ela está dentro de você." Ele entendeu que sua atitude era mais importante do que qualquer ferramenta, qualquer título ou até mesmo de qualquer condição. A partir desse dia, decidiu levar uma vida mais leve, resignificando problemas antigos que passaram a ter menos importância.*

CAPÍTULO 8

# RECURSÃO SEGURA COM O PADRÃO TRAMPOLINE

> *"Sertão é onde o pensamento da gente se forma mais forte do que o lugar. Viver é muito perigoso..."* - Grande Sertão: Veredas

JavaScript é uma linguagem multiparadigma que suporta Orientação a Objetos e Funcional. Não é raro desenvolvedores mais voltados para o paradigma funcional se valerem de **recursão** para solucionar problemas. Aliás, utilizamos recursão ao implementarmos nosso mecanismo de `retry` com Promises. Relembremos o código:

```
// cangaceiro2/public/app/utils/promise-helpers.js
// código anterior omitido

export const retry = (retries, milliseconds, fn) =>
    fn().catch(err => {
        return delay(milliseconds)().then(() =>
            retries > 1
                ? retry(--retries, milliseconds, fn)
                : Promise.reject(err))
    });
```

A própria função `retry` chama a si mesma enquanto houver tentativas restantes para repetir a operação. Não há nada de errado com nosso exemplo, porém vamos criar outro exemplo que não esteja no meio de outras tecnologias, como Promises, para que possamos focar única e exclusivamente em recursão.

Vamos criar o arquivo `cangaceiro2/public/trampoline.html`:

```
<!-- cangaceiro2/public/trampoline.html -->

<!DOCTYPE html>
<html lang="pt">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Trampoline pattern</title>
</head>
<body>
    <script>
        /* escreveremos nosso código aqui */
    </script>
</body>
</html>
```

Vamos declarar a função recursiva `showCountDown`, que exibirá regressivamente todos os números a partir do número fornecido:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->

<script>

    const showCountDown = counter => {
        if (counter < 0) return;
        console.log(counter);
        showCountDown(--counter);
```

```
    };

    showCountDown(3);
</script>

<!-- código posterior omitido -->
```

Uma função recursiva nada mais é do que uma função que chama a si mesma. Todavia, é preciso haver um *leave event* para indicar o fim da recursão, caso contrário, cairemos em um loop infinito. Em nosso caso, a instrução `if (counter < 0) return;` realiza esse papel.

Ao acessarmos a página no navegador, através do seu console verificamos como saída:

```
3
2
1
0
```

O exemplo anterior poderia ser implementado por meio do tradicional loop `while` , sem a necessidade de recursão. Contudo, há problemas computacionais que são resolvidos com mais facilidade com o uso de recursão, por exemplo, o clássico cálculo fatorial ((https://pt.wikipedia.org/wiki/Fatorial), assunto que abordaremos a seguir.

## 8.1 RECURSÃO E CLÁSSICO CÁLCULO FATORIAL

Vamos implementar a função recursiva `factorial` , aquela que realizará o cálculo factorial:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->
```

```
<script>

    // função showCountDown omitida

    const factorial = n => {
        // fatorial de 0 é 1!
        if (n <= 1) return 1
        // return aqui!
        return n * factorial(--n);
    };

    console.log(factorial(3)) // 6;

</script>

<!-- código posterior omitido -->
```

Ainda sobre a nossa função, uma chamada à `factorial(3)` realizará mais duas chamadas à função `factorial` . Mas como isso aparece na *call stack*? Aliás, o que seria essa call stack?

> *A CALL STACK (pilha de chamadas) é composta de uma ou mais **stack frames**. Cada stack frame corresponde a uma chamada para uma função que ainda não terminou, isto é, que ainda não retornou.*

Cada chamada recursiva adiciona uma *stack frame* à *call stack* até chegar ao *leave event* e, a partir desse ponto, cada *stack frame* começa a ser removida à medida que retorna seu resultado. Visualmente temos algo assim:

```
// apenas exemplo, não entra no programa

factorial(1) // 3 chamada
factorial(2) // 2 chamada
```

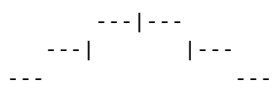
```
factorial(3) // 1 chamada
```

Quando `factorial(1)` for chamado, o *leave event* retornará `1` terminando a chamada recursiva. Isso fará com que cada *stack frame* seja removida da pilha:

```
// exemplo apenas, não entra no programa

factorial(1) // 3 chamada / primeira a ser removida
factorial(2) // 2 chamada / segunda a ser removida
factorial(3) // 1 chamada / terceira a ser removida, retornando o
 fatorial
```

Graficamente temos:



Porém, há um limite máximo de *stack frames* na *call stack* que, ao ser excedido, fará o programa terminar abruptamente. Que tal irmos além da capacidade da *call stack* para vermos o que acontece?

## 8.2 ULTRAPASSANDO O TAMANHO MÁXIMO DA CALL STACK

Vamos voltar ao exemplo do cálculo fatorial, mas desta vez utilizaremos `20000` como argumento da função:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->

<script>

    // função showCountDown omitida

    const factorial = n => {
        if (n <= 1) return 1
```

```
        return n * factorial(--n);
    };

    console.log(factorial(20000));
    // Uncaught RangeError: Maximum call stack size exceeded

</script>

<!-- código posterior omitido -->
```

Recebemos no console a mensagem de erro `Uncaught RangeError: Maximum call stack size exceeded`. Passamos do limite suportado pela *call stack*, isto é, estouramos a pilha de execução do nosso programa!

E agora? Uma solução é adequarmos nossa chamada recursiva para que se beneficie da **Tail Call Optimization (TCO)**.

> Alguns autores chamam de *Tail Call Elimination* (TCE).

Mas o que seria essa tal TCO e como ela pode nos ajudar?

## 8.3 TAIL CALL OPTIMIZATION

A *Tail Call Optimization* (TCO) é um comportamento do interpretador, em nosso caso, de um interpretador JavaScript. Ele ocorre quando o interpretador vê que a chamada de função recorrente é a **última** coisa na função a ser executada e **não há** nenhuma outra operação que precisa ser aplicada ao resultado retornado da função. A chamada para a função corrente é feita via "jump" e não por meio de uma chamada de "sub-rotina" (http://2ality.com/2014/04/call-stack-size.html). Em outras

palavras, o interpretador otimiza a recursão eliminando a última chamada de função da *call stack*.

O exemplo a seguir, que estouraria a *call stack* em uma engine JavaScript **sem** TCO, rodará indefinidamente sem provocar o estouro por uma engine **com** suporte a TCO:

```
// exemplo apenas, não entra ainda em nosso código

const fn = () => fn();
fn();
// -> Engine COM TCO: loop infinito
// -> Engine SEM TCO: Uncaught RangeError: Maximum call stack siz
e exceeded
```

E nossa função `factorial` ? Ela não se coaduna com a exigência da TCO, pois seu retorno é `n * factorial(--n);` e não a chamada de uma função apenas.

A função `factorial` , adequada à TCO fica assim:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->

<script>

    // função showCountDown omitida

    // acc é o acumulador do resultado
    const factorial = (acc, n) => {
        if (n <= 1) return acc;
        // agora retorna a chamada da função
        return factorial(acc * n, --n);
    };

    console.log(factorial(1, 3)) // 6;

</script>

<!-- código posterior omitido -->
```

Tudo muito bonito, só não é perfeito porque as engines JavaScript não suportam TCO! Como assim?

### Sem suporte a TCO nas engines JavaScript

A implementação da TCO faz parte do ES2015 (E6), porém os fabricantes de navegadores a deixaram de lado por questões de segurança que só apareceram durante a sua implementação.

A boa notícia é que, mesmo sem que as `engines` JavaScript suportem TCO, podemos evitar o estouro da *call stack* por meio de recursão com o **pattern Trampoline**.

## 8.4 O PATTERN TRAMPOLINE

O *pattern* Trampoline (trampolim, em português) é um loop que invoca funções que envolve outras funções. Ele é utilizado para que possamos utilizar uma estrutura recursiva em nosso código em interpretadores sem suporte a TCO.

> *Uma função que envolve outra função é chamada de* ***THUNK****.*

A função `trampoline` é implementada por meio de uma função que recebe outra função como argumento. Vamos dar início à sua implementação como primeira função da tag `<script>` de `trampoline.html` . Vamos definir apenas seu esqueleto, que receberá outra função como parâmetro:

```
<!-- cangaceiro2/public/trampoline.html -->
```

```
<!-- código anterior omitido -->

<script>

    // função que recebe outra como argumento

    const trampoline = fn => {
        /* implementação do trampoline */
    };

    // função showCountDown omitida
    // função factorial omitida

</script>

<!-- código posterior omitido -->
```

A função `trampoline` chamará repetidamente a função `fn` recebida como parâmetro até que o retorno da função não seja um `thunk`:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->

<script>

    const trampoline = fn => {

        // faça enquanto for uma função
        while (typeof fn === 'function') {
            // chama a função repetidas vezes
            fn = fn();
        }
        // só retornada quando fn não for uma função!
        return fn;
    };

</script>

<!-- código posterior omitido -->
```

Durante as chamadas sucessivas da função original, se algum valor que não for uma função for retornado, o `trampoline`

parará imediatamente sua execução, retornando o resultado da última chamada da função `fn` recebida como parâmetro.

Implementar a função `trampoline` não é suficiente. A função `factorial` que adequamos à TCO precisa retornar uma função ( `thunk` ) que, ao ser invocada, executará a operação recursiva:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->

<script>

// função trampoline omitida
// função showCountDown omitida

// alterando a função factorial!
const factorial = (acc, num) => {
    if (num <= 1) return acc;
    // retorna uma função que ao ser chamada retorna
    // a chamada de factorial
    return () => factorial(acc * num, --num);
}

</script>

<!-- código posterior omitido -->
```

Por fim, um teste que antes estourava a *call stack*:

```
// infity, sem estourar a pilha
console.log(trampoline(factorial(1, 20000)));
```

> *O ato de converter uma função para suportar Trampoline é chamado de* ***trampolining***

Só nos resta convertemos a função `showCountDown` . Segue o código completo do capítulo:

```
<!-- cangaceiro2/public/trampoline.html -->
<!-- código anterior omitido -->

<script>

    const trampoline = fn => {
        while (typeof fn === 'function') {
            fn = fn();
        }
        return fn;
    };

    const showCountDown = counter => {
        if (counter < 0) return;
        console.log(counter);
        // antes era "return showCountDown(--counter)",
        // agora retorna uma função
        return () => showCountDown(--counter);
    };

    // exibe o contador sem estourar a pilha
    trampoline(showCountDown(20000));

    const factorial = (acc, num) => {
        if (num <= 1) return acc;
        return () => factorial(acc * num, --num);
    }

    console.log(trampoline(factorial(1, 20000)));

</script>

<!-- código posterior omitido -->
```

A função `trampoline` existe apenas para controlar a execução de forma iterativa, garantindo que haja apenas uma *stack frame* em qualquer momento. Todavia, o código não executa tão rápido quanto sua versão recursiva, além de precisarmos ajustar a função para fazer uso do `trampoline`.

## 8.5 CONCLUSÃO

Por mais que a TCO tenha sido proposta no ES2015 (ES6), sua implementação ainda não chegou às engines dos navegadores do mercado por ter criado brechas de segurança, inclusive na plataforma Node.js. Sua ausência pode ser contornada com o *pattern Trampoline*, que exige uma ligeira modificação no código original. Todavia, o uso do *pattern* não é tão performático quanto o suporte nativo à TCO.

Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part3/08

Capítulo 9

# FUNÇÕES VELOZES COM O PADRÃO MEMOIZATION

> *"É melhor escrever errado a coisa certa, do que escrever certo a coisa errada."* - Patativa do Assaré

Durante a construção da nossa aplicação, criamos as funções `debounceTime` e `takeUntil`, que tinham como finalidade limitar operações do usuário, poupando requisições desnecessárias para a API. No entanto, podemos acrescentar mais uma técnica para melhorar o desempenho da aplicação.

Podemos guardar em um cache o retorno de função que desejamos otimizar. Internamente, o valor no cache estará associado aos parâmetros utilizados na invocação da função e, quando a função for executada novamente com mesmos parâmetros, recuperaremos o valor do cache, evitando reprocessamento. Sem dúvidas, é uma otimização para nenhum cangaceiro botar defeito! Em suma, o que queremos fazer é aplicar

o padrão **Memoization**.

*Memoization* é uma técnica de otimização que consiste no cache do resultado de uma função baseada nos parâmetros de entrada. Neste capítulo, veremos como implementá-la na linguagem JavaScript e vamos verificar na prática seus benefícios.

> *Memoization é uma derivação da palavra `memorandum` em latim, que significa "ser lembrado". Alguns chamam de Memoisation Pattern, utilizando a letra s no lugar de z.*

É irresistível usar o exemplo do cálculo fatorial para fins didáticos! Mas primeiro, vamos criar o arquivo `cangaceiro2/public/memoize.html` com a seguinte estrutura:

```
<!-- cangaceiro2/public/memoize.html -->

<!DOCTYPE html>
<html lang="pt">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Memoize pattern</title>
</head>
<body>
    <script>

        const factorial = n => {
            if (n <= 1) return 1
            return n * factorial(--n);
        };

    </script>
</body>
```

```
</html>
```

Podemos acessar a página diretamente no sistema de arquivos ou, caso o servidor esteja de pé, pelo endereço `http://localhost:3000/memoize.html`.

Temos uma função recursiva que não adere à TCO - inclusive podendo estourar a pilha de execução, dependendo do número recebido como parâmetro. Aliás, vimos no capítulo anterior como contornar a ausência de TCO por meio do *pattern Trampoline*.

Voltando para nossa função `factorial`, vamos calcular o fatorial de **5** para logo depois calcularmos o fatorial de **3**:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    const factorial = n => {
        if (n <= 1) return 1
        return n * factorial(--n);
    };

    console.log(factorial(5)); // 120
    console.log(factorial(3)); // 6

</script>
<!-- código posterior omitido -->
```

Excelente, porém uma observação minuciosa verificará que, ao calcularmos o fatorial de 5, já calculamos o fatorial de 3. Que tal se a função `factorial` guardasse em algum lugar os números fatoriais já calculados? Nesse sentido, a chamada de `factorial(3)` buscaria o resultado desse tal lugar, sem ter que calculá-lo. Uma solução para nosso problema é a aplicação do padrão *Memoization*, que vimos na introdução do capítulo.

## 9.1 IMPLEMENTAÇÃO DO PATTERN MEMOIZATION

Para que possamos compreender com clareza o *pattern Memoization*, trabalharemos com um exemplo de escopo menor, uma função que soma dois números:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    const factorial = n => {
        if (n <= 1) return 1
        return n * factorial(--n);
    };

    console.log(factorial(5));
    console.log(factorial(3));

    // nova função
    const sumTwoNumbers = (num1, num2) => num1 + num2;

    console.log(sumTwoNumbers(5, 5)); // 10
    console.log(sumTwoNumbers(7, 2)); // 9
    console.log(sumTwoNumbers(3, 3)); // 6
    // repetiu a operação!
    console.log(sumTwoNumbers(5, 5)); // 10

</script>
<!-- código posterior omitido -->
```

Nossa função `sumTwoNumbers` é uma função **pura** (*pure function*). Funções puras são aquelas que retornam os mesmos valores para os mesmos parâmetros. Essa característica de uma função pura é chamada de *referential transparency*. Excelente, pois para aplicarmos o *Memoization* precisamos ter a garantia de que, ao passarmos os mesmos parâmetros à função, ela retornará sempre o mesmo valor.

Queremos que a última chamada de `sumTwoNumbers(5,5)` não seja calculada, pelo contrário, queremos obter o resultado de um *cache*, pois ele já foi calculado anteriormente. Para isso, vamos implementar a função `memoizer`, aquela que aplicará o *pattern* **Memoization**:

Apesar de ainda não termos implementado a função `memoizer`, quando pronta, ela será utilizada dessa maneira:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    const factorial = n => {
        if (n <= 1) return 1
        return n * factorial(--n);
    };

    console.log(factorial(5));
    console.log(factorial(3));

    const sumTwoNumbers = (num1, num2) => num1 + num2;

    // memoizer ainda não existe!
    const memoizedSumTwoNumbers = memoizer(sumTwoNumbers);

    console.log(memoizedSumTwoNumbers(5, 5));
    console.log(memoizedSumTwoNumbers(7, 2));
    console.log(memoizedSumTwoNumbers(3, 3));

    // a combinação de parâmetros (5,5) já foi
    // usada antes, por isso obterá o valor do
    // cache
    console.log(memoizedSumTwoNumbers(5, 5));

</script>
<!-- código posterior omitido -->
```

Agora que já sabemos o que esperar da nossa função `memoizer`, vamos implementá-la.

## 9.2 IMPLEMENTANDO A FUNÇÃO MEMOIZER

Nossa função `memoizer` receberá uma função ( `fn` ) como parâmetro e retornará outra, aquela que potencialmente poderá ser chamada múltiplas vezes. Todavia, a função retornada precisará receber um número indefinido de parâmetros que será passado para a função ( `fn` ) em que desejamos aplicar o `memoizer` . Conseguimos isso facilmente por meio de *parâmetros REST* do ES2015. Com eles, podemos passar uma quantidade indeterminada de parâmetros que serão referenciados internamente na função como um array.

Vejamos a estrutura inicial que criaremos logo no início da tag `<script>` , antes das demais funções:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    // recebe uma função como parâmetro
    const memoizer = fn => {
        // retorna uma função que recebe
        // um número indefinido de parâmetros
        return (...args) => {
            // usou REST operator para tratar todos
            // os parâmetros como um array
        };
    };

    //  funções omitidas

</script>

<!-- código posterior omitido -->
```

Precisamos de um local para guardar os resultados computados

e para isso utilizaremos um `Map`:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    const memoizer = fn => {
        const cache = new Map();

        return (...args) => {

        };
    };

    // funções omitidas
</script>
<!-- código posterior omitido -->
```

Toda vez que a nova função retornada por `memoizer` for chamada, ela precisará verificar se os parâmetros recebidos já existem em nosso `Map`. Os parâmetros serializados (convertidos em texto) da função representam a `key` do nosso `Map`. Uma vez existindo, retornaremos o valor guardado, evitando a execução da função `fn`. Se não existir, precisamos adicioná-lo em nosso `Map` e retornar o valor computado:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    const memoizer = fn => {
        const cache = new Map();

        return (...args) => {
            // serializa o array args
            const key = JSON.stringify(args);

            // verifica se a chave existe
            if (cache.has(key)) {
```

```
                console.log(`Buscou do cache ${args}`);

                // retorna o valor do cache
                return cache.get(key);
            } else {
                console.log(
                    `Não encontrou no cache ${args}. Adicionando
ao cache.`
                );

                // invoca a função fn com os
                // parâmetros utilizando
                // o spread operator

                const result = fn(...args);

                // guarda o resultado no cache
                cache.set(key, result);

                // retorna o valor que acabou de ser
                // calculado
                return result;
            }
        };
    };

    // depois funções omitidas

</script>
<!-- código posterior omitido -->
```

Temos nossa função pronta. Vamos testá-la?

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    // função memoize omitida
    // função factorial omitida

    const sumTwoNumbers = (num1, num2) => num1 + num2;

    // turbina a função sumTwoNumbers
```

```
    const memoizedSumTwoNumbers = memoizer(sumTwoNumbers);

    console.log(memoizedSumTwoNumbers(5, 5));
    console.log(memoizedSumTwoNumbers(7, 2));
    console.log(memoizedSumTwoNumbers(3, 3));

    // a combinação de parâmetros já foi usada antes
    console.log(memoizedSumTwoNumbers(5, 5)); // busca do cache

</script>

<!-- código posterior omitido -->
```

Visualizando o log das chamadas a `memoizedSumTwoNumbers`, temos como resultado:

```
Não encontrou no cache 5,5. Adicionando ao cache.
10
Não encontrou no cache 7,2. Adicionando ao cache.
9
Não encontrou no cache 3,3. Adicionando ao cache.
6
Buscou do cache 5,5
10
```

Que tal voltarmos para nossa função `factorial`?

## 9.3 MEMOIZING DA FUNÇÃO FACTORIAL

Há um detalhe em nossa função `factorial` devido à sua natureza recursiva. Não podemos simplesmente aplicar a função `memoizer` como fizemos antes:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>
    // função memoize omitida

    const factorial = n => {
        if (n <= 1) return 1
```

```
        return n * factorial(--n);
    };

    const memoizedFactorial = memoizer(factorial);
    console.log('memoizedFactorial');
    console.log(memoizedFactorial(5));
    console.log(memoizedFactorial(3));
    console.log(memoizedFactorial(4));

    // função sumTwoNumbers omitida
    // função getDiscount omitida
</script>

<!-- código posterior omitido -->
```

Será exibido no console do Chrome:

```
Não encontrou no cache 5. Adicionando ao cache.
120
Não encontrou no cache 3. Adicionando ao cache.
6
Não encontrou no cache 4. Adicionando ao cache.
24
```

Apenas o resultado final de `factorial` entrará no cache. Então, quando chamarmos `memoizedFactorial(3)`, o fatorial de 3 que já foi calculado pelo fatorial de 5 não estará no cache, muito menos o fatorial de 4.

Para contornarmos esse problema, vamos declarar nossa função `factorial` com auxílio da função `memoizer`.

A função `memoizer` receberá como parâmetro uma função com a lógica do cálculo fatorial e retornará uma nova função que possui a capacidade de fazer o cache dos resultados. A função `factorial` continua sendo usada da mesma forma, aliás o desenvolvedor nem saberá que ela realiza o cache das operações, apenas ao ver o trecho de código que a cria:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>
    // função memoize omitida


    const factorial = memoizer(
        n => {
            if (n <= 1) return 1;
            return n * factorial(--n);
        }
    );

    console.log('memoizedFactorial');
    console.log(factorial(5));
    console.log(factorial(3));
    console.log(factorial(4));

    // função sumTwoNumbers omitida
    // função getDiscount omitida
</script>

<!-- código posterior omitido -->
```

A saída no console do navegador será:

```
Não encontrou no cache 5. Adicionando ao cache.
Não encontrou no cache 4. Adicionando ao cache.
Não encontrou no cache 3. Adicionando ao cache.
Não encontrou no cache 2. Adicionando ao cache.
Não encontrou no cache 1. Adicionando ao cache.
120
Buscou do cache 3
6
Buscou do cache 4
24
```

Uma ligeira modificação, com um resultado bem interessante.

Com base no que aprendemos, será que podemos realizar o cache do resultado de Promises? É o que veremos a seguir.

## 9.4 MEMOIZATION DE PROMISES

Vamos criar a função `getNotaFromId()` com a finalidade de consumir nossa API e retornar uma nota fiscal dado seu ID. Não se preocupe, o servidor está preparado para responder ao *endpoint* acessado, contanto que o ID da nota exista:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    // função memoize omitida
    // função factorial omitida
    // função sumTwoNumbers omitida

    // como não estamos usando os módulos da aplicação,
    // criamos rapidamente um handler
    const requestHandler = res =>
        res.ok ? res.json() : Promise.reject(res.statusText)

    // retorna uma Promise
    const getNotaFromId = id =>
        fetch(`http://localhost:3000/notas/${id}`)
            .then(requestHandler);

</script>
<!-- código posterior omitido -->
```

Se executarmos a operação duas vezes para um mesmo ID, acessaremos a API duas vezes. Para termos certeza de que uma função executará depois da outra, vamos encadear as duas chamadas:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    // função memoize omitida
    // função factorial omitida
```

```
    // função sumTwoNumbers omitida
    // função getDiscount omitida

    // como não estamos usando os módulos da aplicação,
    // criamos rapidamente um handler
    const requestHandler = res =>
        res.ok ? res.json() : Promise.reject(res.statusText)

    // retorna uma Promise
    const getNotaFromId = id =>
        fetch(`http://localhost:3000/notas/${id}`)
            .then(requestHandler);

    getNotaFromId(1) // busca da API
    .then(console.log)
    .catch(console.log);

</script>
<!-- código posterior omitido -->
```

A boa notícia é que podemos utilizar nossa função `memoizer` para realizar o cache do retorno da Promise:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>
    // função memoize omitida
    // função factorial omitida
    // função sumTwoNumbers omitida
    // função getDiscount omitida

    const requestHandler = res =>
        res.ok ? res.json() : Promise.reject(res.statusText)

    // retorna uma Promise
    const getNotaFromId = id =>
        fetch(`http://localhost:3000/notas/${id}`)
            .then(requestHandler);

    // usando memoizer
    const memoizedGetNotaFromId = memoizer(getNotaFromId);

    memoizedGetNotaFromId(1)
```

```
        .then(console.log)
        .then(() => memoizedGetNotaFromId(1))
        .then(console.log)
        .catch(console.log);
</script>

<!-- código posterior omitido -->
```

Consultando o console do navegador, vemos as seguintes mensagens emitidas pela função `memoizer`:

```
Não encontrou no cache 1. Adicionando ao cache.
Buscou do cache 1
Não encontrou no cache 2. Adicionando ao cache.
```

No entanto, há uma pegadinha. Se por acaso nossa `Promise` falhar por instabilidade na rede ou por qualquer outro problema, a função `memoizedGetNotaFromId()` fará o cache de uma Promise rejeitada.

Queremos permitir que o desenvolvedor, a qualquer momento, apague o cache quando necessário. Para isso, vamos alterar ligeiramente a função `memoizer`, que passará a oferecer a função `clearCache()`. Vamos aproveitar e eliminar as mensagens do console e simplificar ainda mais nossa função:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    const memoizer = fn => {
        const cache = new Map();

        const newFn = (...args) => {
            const key = JSON.stringify(args);
            if (cache.has(key)) return cache.get(key);
            const result = fn(...args);
            cache.set(key, result);
            return result;
```

```
        };

        // pendurando a função clearCache
        // na própria função memoizer
        newFn.clearCache = () => cache.clear();

        return newFn;
    };

    // código posterior omitido

</script>

<!-- código posterior omitido -->
```

Realizando o teste:

```
<!-- cangaceiro2/public/memoize.html -->
<!-- código anterior omitido -->

<script>

    // função memoize omitida
    // função factorial omitida
    // função sumTwoNumbers omitida
    // função getDiscount omitida

    const requestHandler = res =>
        res.ok ? res.json() : Promise.reject(res.statusText)

    // retorna uma Promise
    const getNotaFromId = id =>
        fetch(`http://localhost:3000/notas/${id}`)
            .then(requestHandler);

    const memoizedGetNotaFromId = memoizer(getNotaFromId);
    memoizedGetNotaFromId(1) // busca da API
        .then(console.log)
        .then(() => memoizedGetNotaFromId.clearCache())
        .then(() => memoizedGetNotaFromId(1)) // busca do API!
        .then(console.log)
        .catch(console.log);

</script>
```

```
<!-- código posterior omitido -->
```

Podemos recorrer à função `clearCache` sempre que necessário.

## 9.5 CONCLUSÃO

O *pattern* **Memoization** permite acelerar a execução de funções cacheando seus resultados. Com suporte a *high order functions*, a linguagem JavaScript permite implementar o *pattern* com menos esforço.

E você? O que achou o *pattern* apresentado? Já precisou realizar o cache do resultado de métodos ou funções? Qual estratégia utilizou? Deixe sua opinião para o cangaceiro JavaScript aqui!

> Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part3/09

Capítulo 10

# O PADRÃO DECORATOR

> *Medo, não, mas perdi a vontade de ter coragem.* - Grande Sertão: Veredas

No meu livro *Cangaceiro JavaScript: uma aventura no sertão da programação*, apliquei o padrão de projeto *Decorator* através de Babel. Vamos revisitar a finalidade de um *Decorator* e a utilidade do Babel.

Um *decorator* nada mais é do que um trecho de código isolado aplicável em uma ou mais funções, inclusive em métodos de classe. A lógica a ser aplicada depende do problema a ser resolvido, mas o mais importante é que teremos essa lógica em apenas um lugar, evitando a duplicação de código.

Babel (https://babeljs.io) é um compilador de código JavaScript que permite utilizarmos novos recursos da linguagem antes mesmo de serem suportados pelos navegadores do mercado.

Neste último capítulo, demonstrarei que é possível aplicar o *pattern Decorator* de maneira padronizada e sem o auxílio de um compilador como Babel, utilizando apenas vanilla JavaScript.

Aliás, utilizamos apenas JavaScript sem a utilização de qualquer biblioteca ao longo dos capítulos.

Para evitar que o leitor divida sua atenção com o código que já escrevemos durante o livro e para focarmos exclusivamente na implementação do *Decorator*, criaremos um projeto em separado.

Primeiro, criaremos o módulo `cangaceiro2/public/app/models/person.js`. Nele, definiremos a classe `Person` com as propriedades `_name` e `_surname` e com os métodos `speak` e `getFullName`:

```
// cangaceiro2/app/models/person.js;

export class Person {

    constructor(name, surname) {
        this._name = name;
        this._surname = surname;
    }

    speak(phrase) {
        return `${this._name} is speaking... ${phrase}`
    }

    getFullName() {
        return `${this._name} ${this._surname}`;
    }
}
```

Na prática, poderia ser qualquer classe, contanto que tivesse métodos e que alguns deles recebessem parâmetros. Vamos ao problema.

Não é raro o desenvolvedor precisar medir o tempo de execução de métodos e funções com o objetivo de encontrar problemas de performance. Devido ao baixo grau de complexidade da nossa classe, não teremos problemas de performance, mas a

questão aqui é: como proceder caso queiramos obter métricas dos nossos métodos?

Uma solução é alterarmos os métodos adicionando todo o código necessário para cronometrar o tempo. Conseguimos isso por meio das funções `console.time` e `console.timeEnd`:

```
// app/models/person.js;

export class Person {

    constructor(name, surname) {
        this._name = name;
        this._surname = surname;
    }

    speak(phrase) {
        console.time('speak'); // novidade
        const result = `${this._name} is speaking... ${phrase}`;
        // exibe o tempo gasto
        console.timeEnd('speak');
        return result;
    }

    getFullName() {
        console.time('speak');// novidade
        const result = `${this._name} ${this._surname}`;
         // exibe o tempo gasto
        console.timeEnd('speak');
        return result;
    }
}
```

Vamos criar o módulo `cangaceiro2/public/app/app2.js`, para que possamos testar nosso código dentro da infraestrutura que já temos. Nele, vamos criar uma instância de `Person` para que possamos verificar a saída no console:

```
// cangaceiro2/public/app/app2.js

import { Person } from './models/person.js';
```

```
const person = new Person('Flávio', 'Almeida');
person.getFullName();
person.speak('Canganceiro JavaScript');
```

Por fim, vamos criar a página `cangaceiro2/public/decorator.html`, que importará o módulo `app2.js`. É a partir dela que conseguiremos executar todo o código que escreveremos neste capítulo:

```
<!-- cangaceiro2/public/decorator.html -->

<!DOCTYPE html>
<html lang="pt">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Decorator pattern</title>
</head>
<body>
    <script type="module" src="app/app2.js"></script>
</body>
</html>
```

Como estamos usando o sistema de módulos nativos do navegador, precisamos acessar a página através de `http://localhost:3000/decorator.html`, por isso tenha certeza de que o servidor esteja de pé.

Depois de carregarmos a página no navegador, podemos verificar as informações do tempo de execução dos métodos no console do próprio navegador. Nada especial aqui. Contudo, não precisamos meditar muito para verificarmos que duplicamos nossa lógica que computa o tempo de execução nos métodos `speak` e `getFullName`. Aliás, teríamos ainda mais código duplicado se outras classes da nossa aplicação também precisassem cronometrar

a execução de seus métodos.

Uma solução é isolarmos o código duplicado em um único lugar e aplicá-lo aos métodos da classe. Bingo! Chegamos ao padrão de projeto *Decorator* descrito no início do capítulo. Chegou a hora de implementá-lo, mas antes de colocarmos as mãos na massa, vamos pegar como inspiração o suporte a decorators da linguagem TypeScript.

## 10.1 TYPESCRIPT COMO INSPIRAÇÃO

Em TypeScript, podemos isolar um código e aplicá-lo em métodos de classes por meio da sintaxe `@nomeDoDecorator`, sendo assim, o nome `@logExecutionTime` faz muito sentido para o problema que temos, que é logar o tempo de execução das funções sem termos que repetir código.

O código a seguir apenas demonstra onde o *decorator* `@logExecutionTime` será aplicado e não sua implementação, até porque nosso foco é implementá-lo utilizando apenas os recursos vigentes da linguagem JavaScript:

```
// apenas exemplo, não entra em nosso código

export class Person {

    constructor(name, surname) {
        this._name = name;
        this._surname = surname;
    }

    // aplicação do decorator
    @logExecutionTime
    speak(phrase) {
        return `${this._name} is speaking... ${phrase}`
    }
```

```
    // aplicação do decorator
    @logExecutionTime
    getFullName() {
        return `${this._name} ${this._surname}`;
    }
}
```

Veja que em nenhum momento na classe `Pessoa` vemos a lógica encapsulada pelo *decorator* `@logExecutionTime`, inclusive o mesmo *decorator* é aplicado em métodos diferentes. Será que podemos chegar a um resultado semelhante diretamente na linguagem JavaScript? Antes de tentarmos, vamos remover as instruções `console.time` e `console.timeEnd` dos métodos da classe `Person`, que ficará assim:

```
// app/models/person.js;
export class Person {

    constructor(name, surname) {
        this._name = name;
        this._surname = surname;
    }

    speak(phrase) {
        const result = `${this._name} is speaking... ${phrase}`;
        return result;
    }

    getFullName() {
        const result = `${this._name} ${this._surname}`;
        return result;
    }
}
```

Vamos partir para a primeira solução.

## 10.2 PRIMEIRA SOLUÇÃO

Pela característica dinâmica da linguagem JavaScript, podemos alterar o método `speak` diretamente no prototype de `Person`. Para isso, precisamos guardar o método original para então chamá-lo na nova função que o substituirá. Primeiro, veremos a implementação para depois analisá-la:

```
// cangaceiro2/public/app/app2.js

import { Person } from './models/person.js';

// guardou o método original
const method = Person.prototype.speak;

// substitui o método por uma função
Person.prototype.speak = function (...args) {
    console.log(`Argumentos do método: ${args}`);
    console.time('speak');
    // executa o código original
    const result = method.bind(this)(...args);
    console.log(`Resultado do método: ${result}`);
    console.timeEnd('speak');
    return result;
};

const person = new Person('Flávio', 'Almeida');
person.speak('Canganceiro JavaScript');
```

Vamos analisar o código. Guardamos uma referência para o método definido no prototype da classe `Person` na variável `method`.

> *Não se esqueça de que, ao declararmos uma classe com a sintaxe `class`, as definições de seus métodos são adicionadas em seu prototype. É por isso que acessamos `Person.prototype.speak` e não `Person.speak`.*

Com o método original guardado, substituímos `Person.prototype.speak` por uma função que redefine o método. Ela recebe um número indeterminado de parâmetros por meio do *Rest operator* (os famosos três pontinhos), pois não sabemos quantos parâmetros o método original recebe.

Vale ressaltar que utilizamos uma função no lugar de uma *arrow function* pois necessitamos de um escopo dinâmico, isto é, que o `this` seja a instância que invoca o método naquele momento.

Dentro da função que define o novo método, executamos um código arbitrário antes e depois da chamada do método original. Todavia, um trecho de código merece destaque:

```
// cangaceiro2/public/app/app2.js
// código anterior omitido

// analisando a linha de código
const result = method.bind(this)(...args);

// código posterior omitido
```

Com o `method.bind(this)`, criamos uma nova referência para o método original que tem como contexto o `this` da nova função. Lembre-se de que esse `this` referenciará a instância que estiver invocando o novo método. Em seguida, passamos por meio do *spread operator* cada parâmetro recebido no array `args` como parâmetros individuais para a função.

Executando nosso código, a saída do console será:

```
Método chamado com os parâmetros: Canganceiro JavaScript
Resultado do método: Flávio is speaking... Canganceiro JavaScript
speak: 0.494873046875ms
```

Excelente, mas essa estratégia deixa a desejar. Lembre-se de que precisamos aplicar a mesma lógica ao método `getFullName` e em outras classes quando necessário. Sendo assim, precisamos padronizar nossa solução em algo que seja fácil de usar pelos demais desenvolvedores da equipe.

## 10.3 ISOLANDO DECORATORS E DEFININDO UMA API

Para facilitar a vida do desenvolvedor, queremos a seguinte solução:

```
// cangaceiro2/public/app/app2.js

import { Person } from './models/person.js';

// As demais funções ainda não existem!
// Serão criadas em breve
import { decorate } from './utils/decorate.js';
import { logExecutionTime } from './models/decorators.js';

// decorando os métodos
// speak e getFullName
decorate(Person, {
    speak: logExecutionTime,
    getFullName: logExecutionTime
});

const person = new Person('Flávio', 'Almeida');
person.speak('Canganceiro JavaScript');
```

Na solução que acabamos de ver, temos a função utilitária `decorate` , que receberá dois parâmetros. O primeiro é a classe cujos métodos desejamos decorar. O segundo é o objeto *handler*. **As propriedades do objeto *handler* equivalem aos nomes dos métodos que desejamos decorar na classe**. Seu valor será o *decorator* que desejamos aplicar.

Antes de implementarmos a função `decorate`, vamos implementar a função `logExecutionTime` no novo módulo `cangaceiro2/public/app/models/decorators.js`. É no módulo `decorators` que declararemos todos as nossas funções decoradoras:

```
// cangaceiro2/public/app/models/decorators.js

export const logExecutionTime = (method, property, args) => {
    console.log(`Método decorado: ${property}`);
    console.log(`Argumentos do método ${args}`);
    console.time(property);
    const result = method(...args);
    console.timeEnd(property);
    console.log(`resultado do método: ${result}`)
    return result;
};
```

Temos uma função que recebe três parâmetros, que serão passados pela função utilitária `decorate` que ainda criaremos. Os parâmetros são:

- `method`: o método original a ser decorado, com seu contexto já modificado;
- `property`: o nome do método;
- `args`: os parâmetros que o método recebe (ou não).

O mais interessante é que **definimos uma API para criação de decorators**, isto é, especificamos quais parâmetros qualquer *decorator* que formos criar deverá receber. Essa padronização é importante, pois permitirá que a solução seja utilizada por outros desenvolvedores mais facilmente. Sabendo o que cada parâmetro representa, fica fácil entender a lógica do nosso *decorator*.

A grande questão agora é a implementação da função utilitária `decorate`, centro nervoso da nossa solução.

## 10.4 IMPLEMENTANDO A FUNÇÃO DECORATE

Vamos implementar a função `decorate`. Ela será a responsável por associar um *decorator* ao método de um objeto. Nesse sentido, a única preocupação que o desenvolvedor terá é criar os decorators e não reimplementar a lógica de sua aplicação, papel da função `decorate`.

Vamos criar o módulo `cangaceiro2/public/app/utils/decorate.js`. Nele declararemos a função `decorate`, que receberá a `clazz` e o `handler`:

```
// cangaceiro2/public/app/utils/decorate.js

export const decorate = (clazz, handler) => // falta o resto
```

> *Usamos `clazz` como parâmetro porque a palavra `class` é reservada na linguagem.*

A primeira coisa que faremos é listar todas as chaves do objeto `handler`, pois são elas que indicam qual método decorar da classe. Todavia, só podemos iterar nas propriedades do próprio objeto `handler` (aquelas que adicionamos ao criá-lo) e não do seu prototype, no caso `Object`. Se iterarmos nas propriedades do `prototype`, acabaremos encontrando outras chaves que não dizem respeito aos métodos que desejamos decorar. A função `Object.keys()` nos atende muito bem, pois não itera nas propriedades do prototype:

```
// cangaceiro2/public/app/utils/decorate.js

export const decorate = (target, handler) =>
    // retorna todas as propriedades enumeráveis do objeto
    Object.keys(handler).forEach(property => {
        // falta implementar
    });
```

A cada iteração precisaremos guardar o método original, um dos parâmetros de que nossos *decorators* dependem:

```
// app/utils/decorate.js

export const decorate = (target, handler) =>
    Object.keys(handler).forEach(property => {
        const method = clazz.prototype[property];
    });
```

Armazenamos na variável `method` uma referência para o método que será decorado. Agora precisamos fazer a mesma coisa, mas desta vez para armazenar o *decorator* do método:

```
// app/utils/decorate.js

export const decorate = (target, handler) =>
    Object.keys(handler).forEach(property => {
        const method = clazz.prototype[property];
        const decorator = handler[property];
    });
```

Ótimo! Só precisamos modificar o método original da classe para que invoque nosso *decorator*, lembrando que o *decorator* receberá como parâmetro uma referência para o método original, inclusive já associado ao `this` da função que substituiu o método na classe:

```
// cangaceiro2/public/app/utils/decorate.js

export const decorate = (clazz, handler) => {
    Object.keys(handler).forEach(property => {
        const method = clazz.prototype[property];
```

```
        const decorator = handler[property];

        // Adiciona novo método que ao ser chamado
        // chamará o decorator por debaixo dos panos
        clazz.prototype[property] = function (...args) {
            return decorator(method.bind(this), property, args);
        };
    });
};
```

Excelente, o código que escrevemos até agora é suficiente para que nosso *decorator* seja aplicado. Mas se quisermos aplicar mais de um *decorator* por método?

## 10.5 MÉTODOS COM MAIS DE UM DECORATOR

O *decorator* `logExecutionTime` está com muita responsabilidade. Extrairemos dele o código que loga os dados da função como o seu nome, parâmetros recebidos e seu retorno em um novo *decorator*, que chamaremos de `inspectMethod`:

Alterando `cangaceiro2/public/app/models/decorators.js`:

```
// cangaceiro2/public/app/models/decorators.js

// responsável por medir o tempo de execução apenas
export const logExecutionTime = (method, property, args) => {
    console.time(property);
    const result = method(...args);
    console.timeEnd(property);
    return result;
};

// responsável por logar os parâmetros
export const inspectMethod = (method, property, args) => {
    console.log(`Método decorado: ${property}`);
    console.log(`Argumentos do método ${args}`);
```

```
    const result = method(...args);
    console.log(`resultado do método: ${result}`)
    return result;
};
```

Reparem que nossos *decorators* seguem a mesma API, fantástico!

Agora, alterando `cangaceiro2/public/app/app2.js` para fazer uso do nosso novo *decorator*:

```
// cangaceiro2/public/app/app2.js

import { Person } from './models/person.js';
import { decorate } from './utils/decorate.js';
import { logExecutionTime, inspectMethod } from './models/decorat
ors.js';

// passando um array de decorators
decorate(Person, {
    speak: [inspectMethod, logExecutionTime],
    getFullName: [logExecutionTime]
});

const person = new Person('Flávio', 'Almeida');
// um erro acontecerá!
person.speak('Canganceiro JavaScript');
```

A intenção foi boa, mas recebemos um erro no console:

```
Uncaught TypeError: decorator is not a function
    at Person.clazz.(anonymous function)
```

Infelizmente, a função `decorate` não está preparada para lidar com um `Array` de *decorators*. Precisamos alterá-la para que funcione:

```
// cangaceiro2/public/app/utils/decorate.js

export const decorate = (clazz, handler) => {
    Object.keys(handler).forEach(property => {
        const decorators = handler[property];
```

```
        decorators.forEach(decorator => {
            // o método já pode ter sido decorado antes
            const method = clazz.prototype[property];
            clazz.prototype[property] = function (...args) {
                return decorator(method.bind(this), property, arg
s);
            };
        });
    });
};
```

Excelente, agora podemos combinar *decorators* em um mesmo método.

## 10.6 ORDEM DOS DECORATORS

Se analisarmos atentamente a aplicação dos *decorators*, veremos que são aplicados da direita para a esquerda. Para facilitar o entendimento, vamos alterar nosso código para que o primeiro *decorator* da lista seja o primeiro a ser aplicado, e assim por diante. Conseguimos isso facilmente com uma chamada à função `Array.reverse`:

```
// cangaceiro2/public/app/util/decorate.js

export const decorate = (clazz, handler) => {
    Object.keys(handler).forEach(property => {
        // faz reverse
        const decorators = handler[property].reverse();
        decorators.forEach(decorator => {
            const method = clazz.prototype[property];
            clazz.prototype[property] = function (...args) {
                return decorator(method.bind(this), property, arg
s);
            };
        });
    });
};
```

Agora, vamos alterar a ordem dos *decorators* em `app/app.js` :

```
// cangaceiro2/public/app/app2.js

// código anterior omitido

// agora aplicará da esquerda para a direita
decorate(Person, {
    speak: [logExecutionTime, inspectMethod],
    getFullName: [logExecutionTime]
});

// código posterior omitido
```

Agora fica mais fácil para quem lê o código entender qual a ordem de aplicação dos *decorators*.

Tudo está excelente, mas e se nosso *decorator* precisar receber parâmetros?

## 10.7 DECORATORS QUE RECEBEM PARÂMETROS

Criamos o *decorator* `inspectMethod` , mas nem sempre queremos logar o resultado do método, apenas seus parâmetros. Podemos atender a este requisito facilmente. Primeiro, vamos alterar o *decorator* `inspectMethod` :

```
// cangaceiro2/public/app/models/decorators.js
// código anterior omitido

// função agora recebe um parâmetro
export const inspectMethod = ({ excludeReturn } = {}) =>
    (method, property, args) => {
        console.log(`Método decorado: ${property}`);
        console.log(`Argumentos do método ${args}`);
        const result = method(...args);
        // só loga se excludeReturn é false
        if(!excludeReturn) console.log(`resultado do método: ${re
```

```
sult}`)
        return result;
    };
```

Vejam que nosso *decorator* recebe como parâmetro um objeto JavaScript que, ao sofrer o *destructuring*, disponibilizará a variável `excludeReturn`, utilizada para controlar a exibição da variável `result`. Por fim, o retorno do *decorator* será a função que adere à API que define nossos *decorators*.

Alterando o módulo `app2.js`:

```
// cangaceiro2/public/app/app2.js

import { Person } from './models/person.js';
import { decorate } from './utils/decorate.js';
import { logExecutionTime, inspectMethod } from './models/decorat
ors.js';

decorate(Person, {
    speak: [inspectMethod({ excludeReturn: true }), logExecutionT
ime],
    getFullName: [logExecutionTime]
});

const person = new Person('Flávio', 'Almeida');
person.speak('Canganceiro JavaScript');
person.getFullName();
```

Se não passarmos parâmetro nenhum para `inspectMethod`, o padrão continuará sendo inspecionar o retorno do método, excelente.

Poderíamos dar como pronto nossa implementação do *pattern Decorator*, mas podemos torná-la ainda melhor. Por exemplo, hoje precisamos decorar cada instância de uma classe, ou seja, se tivermos 200 objetos criados em série precisaremos aplicar a função `decorate` em todos eles.

Que tal decorarmos diretamente os métodos da classe? Com essa abordagem, todas as instâncias criadas a partir dessa classe já terão seus métodos decorados. É essa mudança que veremos a seguir.

## 10.8 APLICANDO O DECORATOR DIRETAMENTE NA DEFINIÇÃO DA CLASSE

Para que possamos garantir que todas as instâncias de `Person` já tenham seus métodos decorados, vamos alterar `app/models/person.js` e aplicar a função `decorate` logo após a definição da classe:

```
// cangaceiro2/app/models/person.js;

// importou a função utilitária e os decorators
import { decorate } from '../utils/decorate.js';
import { logExecutionTime, inspectMethod } from './decorators.js'
;

export class Person {

    constructor(name, surname) {
        this._name = name;
        this._surname = surname;
    }

    speak(phrase) {
        return `${this._name} is speaking... ${phrase}`
    }

    getFullName() {
        return `${this._name} ${this._surname}`;
    }
}

// logo após a definição da classe, decorou os métodos
decorate(Person, {
    speak: [
```

```
        inspectMethod({ excludeReturn: true }),
        logExecutionTime
    ],
    getFullName: [logExecutionTime]
});
```

Por fim, nosso nódulo `app2.js` ficará desta forma:

```
// cangaceiro2/public/app/app2.js

import { Person } from './models/person.js';

const person = new Person('Flávio', 'Almeida');
person.speak('Canganceiro JavaScript');
person.getFullName();
```

Nossos *decorators* continuam sendo aplicados, mas desta vez serão aplicados para qualquer instância da classe `Person`, livrando do desenvolvedor a responsabilidade de aplicar o *decorator* por instância.

## 10.9 CONSIDERAÇÕES FINAIS

Assimilamos gradativamente jargões da Programação Funcional aplicando-os quando necessário. Trabalhamos com mônadas, composição, padrões do projeto visando melhorar tanto a experiência do desenvolvedor quanto do usuário. Se Lampião fosse um programador, com certeza ele ficaria abestado com a quantidade de conhecimento adquirido.

O autor deseja que os conhecimentos adquiridos nesta obra ajudem o leitor, seja no seu dia a dia ou quando ele concorrer a uma vaga de emprego que demande tal conhecimento. É impossível prever o que espera o leitor pelo sertão da programação, mas vale uma última dica: quem sabe faz a hora e não espera acontecer.

Para todos os cangaceiros JavaScript, meus votos de sucesso.

Você encontra o código completo deste capítulo em https://github.com/flaviohenriquealmeida/retorno-cangaceiro-javascript/tree/master/part3/10